# INTRODUCTION TO PYTHON PROGRAMMING LANGUAGE

Most of the slides are from Caitlin Carnahan, Florida State University

## PHILOSOPHY

From *The Zen of Python* (https://www.python.org/dev/peps/pep-0020/)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

2

# NOTABLE FEATURES

- Easy to learn.
- Supports quick development.
- Cross-platform.
- Open Source.
- Extensible.
- Embeddable.
- Large standard library and active community.
- Useful for a wide variety of applications.

3

# INTERPRETER

- The standard implementation of Python is interpreted.
  - You can find info on various implementations here.
- The interpreter translates Python code into bytecode, and this bytecode is executed by the Python VM (similar to Java).
- Two modes: normal and interactive.
  - Normal mode: entire .py files are provided to the interpreter.
  - Interactive mode: read-eval-print loop (REPL) executes statements piecewise.

4

# INTERPRETER: NORMAL MODE

Let's write our first Python program!

In our favorite editor, let's create helloworld.py with the following contents:

```
print "Hello, World!"
```

From the terminal:

```
$ python helloworld.py
Hello, World!
```

Note: In Python 2.x, print is a statement. In Python 3.x, it is a function. If you want to get into the 3.x habit, include at the beginning:

```
from __future__ import print_function
```

Now, you can write

```
print("Hello, World!")
```

5

# INTERPRETER: NORMAL MODE

Let's include a she-bang in the beginning of helloworld.py:

```
#!/usr/bin/env python
print "Hello, World!"
```

Now, from the terminal:

```
$ ./helloworld.py
Hello, World!
```

6

# INTERPRETER: INTERACTIVE MODE

Let's accomplish the same task (and more) in interactive mode (in Linux/Unix environment).

Some options:
-c : executes single command.
-O: use basic optimizations.
-d: debugging info.
More can be found here.

```
$ python
>>> print("Hello, World!")
Hello, World!
>>> hellostring = "Hello, World!"
>>> hellostring
'Hello, World!'
>>> 2*5
10
>>> 2*hellostring
'Hello, World!Hello, World!'
>>> for i in range(0,3):
...     Print("Hello, World!")
...
Hello, World!
Hello, World!
Hello, World!
>>> exit()
$
```

7

# SOME FUNDAMENTALS

- Whitespace is significant in Python. Where other languages may use {} or (), Python uses indentation to denote code blocks.

- Comments
  - Single-line comments denoted by #.
  - Multi-line comments begin and end with three "s.
  - Typically, multi-line comments are meant for documentation.
  - Comments should express information that cannot be expressed in code - do not restate code.

```
# here's a comment
for i in range(0,3):
    print i
def myfunc():
    """here's a comment about
    the myfunc function"""
    print "I'm in a function!"
```

8

# PYTHON TYPING

- Python is a strongly, dynamically typed language.
- Strong Typing
  - Obviously, Python isn't performing static type checking, but it does prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
  - Example: `2 + "four"` ← not going to fly
- Dynamic Typing
  - All type checking is done at runtime.
  - No need to declare a variable or give it a type before use.

Let's start by looking at Python's built-in data types.

9

# NUMERIC TYPES

The subtypes are int, long, float and complex.

- Their respective constructors are int(), long(), float(), and complex().
- All numeric types, except complex, support the typical numeric operations you'd expect to find (a list is available here).
- Mixed arithmetic is supported, with the "narrower" type widened to that of the other. The same rule is used for mixed comparisons.

10

# NUMERIC TYPES

- Numeric
  - **int**: equivalent to C's long int in 2.x but unlimited in 3.x.
  - **float**: equivalent to C's doubles.
  - **long**: unlimited in 2.x and unavailable in 3.x.
  - **complex**: complex numbers.
- Supported operations include constructors (i.e. int(3)), arithmetic, negation, modulus, absolute value, exponentiation, etc.

```
$ python
>>> 3 + 2
5
>>> 18 % 5
3
>>> abs(-7)
7
>>> float(9)
9.0
>>> int(5.3)
5
>>> complex(1,2)
(1+2j)
>>> 2 ** 8
256
```

11

# SEQUENCE DATA TYPES

There are seven sequence subtypes: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

All data types support arrays of objects but with varying limitations.

The most commonly used sequence data types are strings, lists, and tuples. The xrange data type finds common use in the construction of enumeration-controlled loops. The others are used less commonly.

12

# SEQUENCE TYPES: STRINGS

Created by simply enclosing characters in either single- or double-quotes.

It's enough to simply assign the string to a variable.

Strings are immutable.

There are a tremendous amount of built-in string methods (listed here).

```
mystring = "Hi, I'm a string!"
```

13

# SEQUENCE TYPES: STRINGS

Python supports a number of escape sequences such as '\t', '\n', etc.

Placing 'r' before a string will yield its raw value.

There is a string formatting operator '%' similar to C. A list of string formatting symbols is available here.

Two string literals beside one another are automatically concatenated together.

```
print "\tHello,\n"
print r"\tWorld!\n"
print "Python is " "so cool."
```

```
$ python ex.py
        Hello,

\tWorld!\n
Python is so cool.
```

14

# SEQUENCE TYPES: UNICODE STRINGS

- Unicode strings can be used to store and manipulate Unicode data.
- As simple as creating a normal string (just put a 'u' on it!).
- Use Unicode-Escape encoding for special characters.
- Also has a raw mode, use 'ur' as a prefix.
- To translate to a regular string, use the .encode() method.
- To translate from a regular string to Unicode, use the unicode() function.

```
myunicodestr1 = u"Hi Class!"
myunicodestr2 = u"Hi\u0020Class!"
print myunicodestr1, myunicodestr2
newunicode = u'\xe4\xf6\xfc'
print newunicode
newstr = newunicode.encode('utf-8')
print newstr
print unicode(newstr, 'utf-8')
```

Output:
Hi Class! Hi Class!
äöü
äöü
äöü

15

# SEQUENCE TYPES: LISTS

- Lists are an incredibly useful *compound* data type.
- Lists can be initialized by the constructor, or with a bracket structure containing 0 or more elements.
- Lists are mutable – it is possible to change their contents. They contain the additional mutable operations.
- Lists are nestable. Feel free to create lists of lists…

```
mylist = [42, 'apple', u'unicode apple', 5234656]
print mylist
mylist[2] = 'banana'
print mylist
mylist[3] = [['item1', 'item2'], ['item3', 'item4']]
print mylist
mylist.sort()
print mylist
print mylist.pop()
mynewlist = [x*2 for x in range(0,5)]
print mynewlist
```

[42, 'apple', u'unicode apple', 5234656]
[42, 'apple', 'banana', 5234656]
[42, 'apple', 'banana', [['item1', 'item2'], ['item3', 'item4']]]
[42, [['item1', 'item2'], ['item3', 'item4']], 'apple', 'banana']
banana
[0, 2, 4, 6, 8]

16

# SEQUENCE DATA TYPES

- Sequence
  - **str**: string, represented as a sequence of 8-bit characters in Python 2.x.
  - **unicode**: stores an abstract sequence of code points.
  - **list**: a compound, mutable data type that can hold items of varying types.
  - **tuple**: a compound, immutable data type that can hold items of varying types. Comma separated items surrounded by parentheses.
  - a few more – we'll cover them later.

```
$ python
>>> mylist = ["spam", "eggs", "toast"] # List of strings!
>>> "eggs" in mylist
True
>>> len(mylist)
3
>>> mynewlist = ["coffee", "tea"]
>>> mylist + mynewlist
['spam', 'eggs', 'toast', 'coffee', 'tea']
>>> mytuple = tuple(mynewlist)
>>> mytuple
('coffee', 'tea')
>>> mytuple.index("tea")
1
>>> mylonglist = ['spam', 'eggs', 'toast', 'coffee', 'tea']
>>> mylonglist[2:4]
['toast', 'coffee']
```

17

# COMMON SEQUENCE OPERATIONS

All sequence data types support the following operations.

| Operation | Result |
|---|---|
| `x in s` | True if an item of s is equal to x, else False. |
| `x not in s` | False if an item of s is equal to x, else True. |
| `s + t` | The concatenation of s and t. |
| `s * n, n * s` | n shallow copies of s concatenated. |
| `s[i]` | ith item of s, origin 0. |
| `s[i:j]` | Slice of s from i to j. |
| `s[i:j:k]` | Slice of s from i to j with step k. |
| `len(s)` | Length of s. |
| `min(s)` | Smallest item of s. |
| `max(s)` | Largest item of s. |
| `s.index(x)` | Index of the first occurrence of x in s. |

# COMMON SEQUENCE OPERATIONS

Mutable sequence types further support the following operations.

| Operation | Result |
|---|---|
| `s[i] = x` | Item i of s is replaced by x. |
| `s[i:j] = t` | Slice of s from i to j is replaced by the contents of t. |
| `del s[i:j]` | Same as s[i:j] = []. |
| `s[i:j:k] = t` | The elements of s[i:j:k] are replaced by those of t. |
| `del s[i:j:k]` | Removes the elements of s[i:j:k] from the list. |
| `s.append(x)` | Add x to the end of s. |

19

# COMMON SEQUENCE OPERATIONS

Mutable sequence types further support the following operations.

| | |
|---|---|
| `s.extend(x)` | Appends the contents of x to s. |
| `s.count(x)` | Return number of i's for which s[i] == x. |
| `s.index(x[, i[, j]])` | Return smallest k such that s[k] == x and i <= k < j. |
| `s.insert(i, x)` | Insert x at position i. |
| `s.pop([i])` | Same as x = s[i]; del s[i]; return x. |
| `s.remove(x)` | Same as del s[s.index(x)]. |
| `s.reverse()` | Reverses the items of s in place. |
| `s.sort([cmp[, key[, reverse]]])` | Sort the items of s in place. |

20

# BASIC BUILT-IN DATA TYPES

- Set
  - **set**: an unordered collection of unique objects.
  - **frozenset**: an immutable version of set.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange']
>>> fruit = set(basket)
>>> fruit
set(['orange', 'pear', 'apple'])
>>> 'orange' in fruit
True
>>> 'crabgrass' in fruit
False
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b
set(['r', 'd', 'b'])
>>> a | b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
```

21

# BASIC BUILT-IN DATA TYPES

```
>>> gradebook = dict()
>>> gradebook['Susan Student'] = 87.0
>>> gradebook
{'Susan Student': 87.0}
>>> gradebook['Peter Pupil'] = 94.0
>>> gradebook.keys()
['Peter Pupil', 'Susan Student']
>>> gradebook.values()
[94.0, 87.0]
>>> gradebook.has_key('Tina Tenderfoot')
False
>>> gradebook['Tina Tenderfoot'] = 99.9
>>> gradebook
{'Peter Pupil': 94.0, 'Susan Student': 87.0, 'Tina Tenderfoot': 99.9}
>>> gradebook['Tina Tenderfoot'] = [99.9, 95.7]
>>> gradebook
{'Peter Pupil': 94.0, 'Susan Student': 87.0, 'Tina Tenderfoot': [99.9, 95.7]}
```

- Mapping
  - **dict**: hash tables, maps a set of keys to arbitrary objects.

22

# CONTROL FLOW TOOLS

While loops have the following general structure.

```
while expression:
    statements
```

Here, *statements* refers to one or more lines of Python code. The conditional expression may be any expression, where any non-zero value is true. The loop iterates while the expression is true.

Note: All the statements indented by the same amount after a programming construct are considered to be part of a single block of code.

```
i = 1
while i < 4:
    print i
    i = i + 1
flag = True
while flag and i < 8:
    print flag, i
    i = i + 1
```

```
1
2
3
True 4
True 5
True 6
True 7
```



# CONTROL FLOW TOOLS

The if statement has the following general form.

```
if expression:
    statements
```

If the boolean expression evaluates to True, the statements are executed. Otherwise, they are skipped entirely.

```
a = 1
b = 0
if a:
    print "a is true!"
if not b:
    print "b is false!"
if a and b:
    print "a and b are true!"
if a or b:
    print "a or b is true!"
```

```
a is true!
b is false!
a or b is true!
```

# CONTROL FLOW TOOLS

You can also pair an else with an if statement.

```
if expression:
    statements
else:
    statements
```

The elif keyword can be used to specify an else if statement.

Furthermore, if statements may be nested within eachother.

```
a = 1
b = 0
c = 2
if a > b:
    if a > c:
        print "a is greatest"
    else:
        print "c is greatest"
elif b > c:
    print "b is greatest"
else:
    print "c is greatest"
```

---

c is greatest

25

# CONTROL FLOW TOOLS

The for loop has the following general form.

```
for var in sequence:
    statements
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *var*. Next, the statements are executed. Each item in the sequence is assigned to *var*, and the statements are executed until the entire sequence is exhausted.

For loops may be nested with other control flow tools such as while loops and if statements.

```
for letter in "aeiou":
    print "vowel: ", letter
for i in [1,2,3]:
    print i
for i in range(0,3):
    print i
```

---

vowel: a
vowel: e
vowel: i
vowel: o
vowel: u
1
2
3
0
1
2

26

# CONTROL FLOW TOOLS

Python has two handy functions for creating a range of integers, typically used in for loops. These functions are range() and xrange().

They both create a sequence of integers, but range() creates a list while xrange() creates an xrange object.

Essentially, range() creates the list statically while xrange() will generate items in the list as they are needed. We will explore this concept further in just a week or two.

For very large ranges – say one billion values – you should use xrange() instead. For small ranges, it doesn't matter.

```
for i in xrange(0, 4):
    print i
for i in range(0,8,2):
    print i
for i in range(20,14,-2):
    print i
```

```
0
1
2
3
0
2
4
6
20
18
16
```

27

# CONTROL FLOW TOOLS

There are four statements provided for manipulating loop structures. These are break, continue, pass, and else.

- break: terminates the current loop.
- continue: immediately begin the next iteration of the loop.
- pass: do nothing. Use when a statement is required syntactically.
- else: represents a set of statements that should execute when a loop terminates.

```
for num in range(10,20):
    if num%2 == 0:
        continue
    for i in range(3,num):
        if num%i == 0:
            break
    else:
        print num, 'is a prime number'
```

```
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
```

28

# A FIBONACCI SOLUTION USING BASIC PYTHON

```
total = 0

f1, f2 = 1, 2

while f1 < 4000000:
    if f1 % 2 == 0:
        total = total + f1
    f1, f2 = f2, f1 + f2

print(total)
```

Notice we're using the Python 3.x version of print here.

Python supports multiple assignment at once.
Right hand side is fully evaluated
before setting the variables.

Output: 4613732

29

# FUNCTIONS

A function is created with the def keyword. The statements in the block of the function must be indented.

```
def function_name(args):
    statements
```

The def keyword is followed by the function name with round brackets enclosing the arguments and a colon. The indented statements form a body of the function.

The return keyword is used to specify a list of values to be returned.

```
# Defining the function
def print_greeting():
    print "Hello!"
    print "How are you today?"

print_greeting() # Calling the function
```

Hello!
How are you today?

30

# CODING STYLE

So now that we know how to write a Python program, let's break for a bit to think about our coding style. Python has a style guide that is useful to follow, you can read about PEP 8 here.

I encourage you all to check out pylint, a Python source code analyzer that helps you maintain good coding standards.

31

# FUNCTIONS

Redundant calls can become a little cumbersome, especially if one of the arguments is likely to have the same value for every call.

## Default argument values

- We can provide a default value for any number of arguments in a function.
- Allows functions to be called with a variable number of arguments.
- Arguments with default values must appear at the end of the arguments list!

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

32

# FUNCTIONS

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

Now we can provide a variable number of arguments. All of the following calls are valid:

- `connect('admin', 'ilovecats')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 6379)`

33

# LAMBDA FUNCTIONS

One can also define lambda functions within Python.

- Use the keyword *lambda* instead of *def*.
- Can be used wherever function objects are used.
- Restricted to one expression.
- Typically used with functional programming tools – we will see this next time.

```
>>> def f(x):
...     return x**2
...
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

34

# LIST COMPREHENSIONS

List comprehensions provide a nice way to construct lists where the items are the result of some operation.

The simplest form of a list comprehension is

```
[expr for x in sequence]
```

Any number of additional for and/or if statements can follow the initial for statement. A simple example of creating a list of squares:

```
>>> squares = [x**2 for x in range(0,11)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

35

# LIST COMPREHENSIONS

Here's a more complicated example which creates a list of tuples.

```
>>> squares = [(x, x**2, x**3) for x in range(0,9) if x % 2 == 0]
>>> squares
[(0, 0, 0), (2, 4, 8), (4, 16, 64), (6, 36, 216), (8, 64, 512)]
```

The initial expression in the list comprehension can be anything, even another list comprehension.

```
>>> [[x*y for x in range(1,5)] for y in range(1,5)]
[[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
```

36

# MORE DATA STRUCTURES

- Lists
  - Slicing
  - Stacks and Queues
- Tuples
- Sets and Frozensets
- Dictionaries
- How to choose a data structure.
- Collections
  - Deques and OrderedDicts

37

# WHEN TO USE LISTS

- When you need a collection of elements of varying type.
- When you need the ability to order your elements.
- When you need the ability to modify or add to the collection.
- When you don't require elements to be indexed by a custom value.
- When you need a stack or a queue.
- When your elements are not necessarily unique.

38

# CREATING LISTS

To create a list in Python, we can use bracket notation to either create an empty list or an initialized list.

```
mylist1 = [] # Creates an empty list
mylist2 = [expression1, expression2, ...]
mylist3 = [expression for variable in sequence]
```

The first two are referred to as *list displays*, where the last example is a *list comprehension*.

39

# CREATING LISTS

We can also use the built-in list constructor to create a new list.

```
mylist1 = list()
mylist2 = list(sequence)
mylist3 = list(expression for variable in sequence)
```

The sequence argument in the second example can be any kind of sequence object or iterable. If another list is passed in, this will create a copy of the argument list.

40

# CREATING LISTS

Note that you cannot create a new list through assignment.

```
# mylist1 and mylist2 point to the same list
mylist1 = mylist2 = []

# mylist3 and mylist4 point to the same list
mylist3 = []
mylist4 = mylist3

mylist5 = []; mylist6 = [] # different lists
```

41

# ACCESSING LIST ELEMENTS

If the index of the desired element is known, you can simply use bracket notation to index into the list.

```
>>> mylist = [34,67,45,29]
>>> mylist[2]
45
```

If the index is not known, use the `index()` method to find the first index of an item. An exception will be raised if the item cannot be found.

```
>>> mylist = [34,67,45,29]
>>> mylist.index(67)
1
```

42

# SLICING AND SLIDING

- The length of the list is accessible through `len(mylist)`.
- Slicing is an extended version of the indexing operator and can be used to grab sublists.

```
mylist[start:end]  # items start to end-1
mylist[start:]     # items start to end of the array
mylist[:end]       # items from beginning to end-1
mylist[:]          # a copy of the whole array
```

- You may also provide a step argument with any of the slicing constructions above.

```
mylist[start:end:step]  # start to end-1, by step
```

43

# SLICING AND SLIDING

- The start or end arguments may be negative numbers, indicating a count from the end of the array rather than the beginning. This applies to the indexing operator.

```
mylist[-1]      # last item in the array
mylist[-2:]     # last two items in the array
mylist[:-2]     # everything except the last two items
```

- Some examples:

```
mylist = [34, 56, 29, 73, 19, 62]
mylist[-2]      # yields 19
mylist[-4::2]   # yields [29, 19]
```

44

# INSERTING/REMOVING ELEMENTS

- To add an element to an existing list, use the append() method.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.append(47)
>>> mylist
[34, 56, 29, 73, 19, 62, 47]
```

- Use the extend() method to add all of the items from another list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.extend([47,81])
>>> mylist
[34, 56, 29, 73, 19, 62, 47, 81]
```

45

# INSERTING/REMOVING ELEMENTS

- Use the insert(*pos, item*) method to insert an item at the given position. You may also use negative indexing to indicate the position.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.insert(2,47)
>>> mylist
[34, 56, 47, 29, 73, 19, 62]
```

- Use the remove() method to remove the first occurrence of a given item. An exception will be raised if there is no matching item in the list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.remove(29)
>>> mylist
[34, 56, 73, 19, 62]
```

46

# WHEN TO USE SETS

- When the elements must be unique.
- When you need to be able to modify or add to the collection.
- When you need support for mathematical set operations.
- When you don't need to store nested lists, sets, or dictionaries as elements.

47

# CREATING SETS

- Create an empty set with the set constructor.

```
myset = set()
myset2 = set([])  # both are empty sets
```

- Create an initialized set with the set constructor or the { } notation. Do not use empty curly braces to create an empty set – you'll get an empty dictionary instead.

```
myset = set(sequence)
myset2 = {expression for variable in sequence}
```

48

# HASHABLE ITEMS

The way a set detects non-unique elements is by indexing the data in memory, creating a hash for each element. This means that all elements in a set must be *hashable*.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are.

Objects which are instances of user-defined classes are also hashable by default.

49

# SET OPERATIONS

- The following operations are available for both set and frozenset types.
- Comparison operators >=, <= test whether a set is a superset or subset, respectively, of some other set. The > and < operators check for proper supersets/subsets.

```
>>> s1 = set('abracadabra')
>>> s2 = set('bard')
>>> s1 >= s2
True
>>> s1 > s2
True
>>> s1 <= s2
False
```

50

# SET OPERATIONS

- Union: `set | other | …`
  - Return a new set with elements from the set and all others.
- Intersection: `set & other & …`
  - Return a new set with elements common to the set and all others.
- Difference: `set - other - …`
  - Return a new set with elements in the set that are not in the others.
- Symmetric Difference: `set ^ other`
  - Return a new set with elements in either the set or other but not both.

51

# SET OPERATIONS

```
>>> s1 = set('abracadabra')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2 = set('alacazam')
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 | s2
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 & s2
set(['a', 'c'])
>>> s1 - s2
set(['b', 'r', 'd'])
>>> s1 ^ s2
set(['b', 'r', 'd', 'l', 'z', 'm'])
```

52

# OTHER OPERATIONS

- `s.copy()` returns a shallow copy of the set *s*.
- `s.isdisjoint(other)` returns True if set *s* has no elements in common with set *other*.
- `s.issubset(other)` returns True if set *s* is a subset of set *other*.
- `len`, `in`, and `not in` are also supported.

53

# WHEN TO USE TUPLES

- When storing elements that will not need to be changed.
- When performance is a concern.
- When you want to store your data in logical immutable pairs, triples, etc.

54

# CONSTRUCTING TUPLES

- An empty tuple can be created with an empty set of parentheses.
- Pass a sequence type object into the tuple() constructor.
- Tuples can be initialized by listing comma-separated values. These do not need to be in parentheses but they can be.
- One quirk: to initialize a tuple with a single value, use a trailing comma.

```
>>> t1 = (1, 2, 3, 4)
>>> t2 = "a", "b", "c", "d"
>>> t3 = ()
>>> t4 = ("red", )
```

55

# TUPLE OPERATIONS

Tuples are very similar to lists and support a lot of the same operations.

- Accessing elements: use bracket notation (e.g. `t1[2]`) and slicing.
- Use `len(t1)` to obtain the length of a tuple.
- The universal immutable sequence type operations are all supported by tuples.
  - +, *
  - in, not in
  - min(t), max(t), t.index(x), t.count(x)

56

# PACKING/UNPACKING

Tuple packing is used to "pack" a collection of items into a tuple. We can unpack a tuple using Python's multiple assignment feature.

```
>>> s = "Susan", 19, "CS" # tuple packing
>>> name, age, major = s # tuple unpacking
>>> name
'Susan'
>>> age
19
>>> major
'CS'
```

57

# WHEN TO USE DICTIONARIES

- When you need to create associations in the form of key:value pairs.
- When you need fast lookup for your data, based on a custom key.
- When you need to modify or add to your key:value pairs.

58

# CONSTRUCTING A DICTIONARY

- Create an empty dictionary with empty curly braces or the dict() constructor.
- You can initialize a dictionary by specifying each key:value pair within the curly braces.
- Note that keys must be *hashable* objects.

```
>>> d1 = {}
>>> d2 = dict() # both empty
>>> d3 = {"Name": "Susan", "Age": 19, "Major": "CS"}
>>> d4 = dict(Name="Susan", Age=19, Major="CS")
>>> d5 = dict(zip(['Name', 'Age', 'Major'], ["Susan", 19, "CS"]))
>>> d6 = dict([('Age', 19), ('Name', "Susan"), ('Major', "CS")])
```

Note: zip takes two equal-length collections and merges their corresponding elements into tuples.

59

# ACCESSING THE DICTIONARY

To access a dictionary, simply index the dictionary by the key to obtain the value. An exception will be raised if the key is not in the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age']
19
>>> d1['Name']
'Susan'
```

60

# UPDATING A DICTIONARY

Simply assign a key:value pair to modify it or add a new pair. The del keyword can be used to delete a single key:value pair or the whole dictionary. The clear() method will clear the contents of the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age'] = 21
>>> d1['Year'] = "Junior"
>>> d1
{'Age': 21, 'Name': 'Susan', 'Major': 'CS', 'Year': 'Junior'}
>>> del d1['Major']
>>> d1
{'Age': 21, 'Name': 'Susan', 'Year': 'Junior'}
>>> d1.clear()
>>> d1
{}
```

61

# BUILT-IN DICTIONARY METHODS

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1.has_key('Age') # True if key exists
True
>>> d1.has_key('Year') # False otherwise
False
>>> d1.keys() # Return a list of keys
['Age', 'Name', 'Major']
>>> d1.items() # Return a list of key:value pairs
[('Age', 19), ('Name', 'Susan'), ('Major', 'CS')]
>>> d1.values() # Returns a list of values
[19, 'Susan', 'CS']
```

Note: `in`, `not in`, `pop(key)`, and `popitem()` are also supported.

62

# ORDERED DICTIONARY

Dictionaries do not remember the order in which keys were inserted. An ordered dictionary implementation is available in the collections module. The methods of a regular dictionary are all supported by the OrderedDict class.

An additional method supported by OrderedDict is the following:

```
OrderedDict.popitem(last=True)  # pops items in LIFO order
```

63

# EXCEPTIONS

Errors that are encountered during the execution of a Python program are *exceptions.*

```
>>> print spam
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

There are a number of built-in exceptions, which are listed here.

64

# HANDLING EXCEPTIONS

Explicitly handling exceptions allows us to control otherwise undefined behavior in our program, as well as alert users to errors. Use try/except blocks to catch and recover from exceptions.

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...
Enter a number: two
Ooops !! That was not a valid number. Try again.
Enter a number: 100
```

65

# HANDLING EXCEPTIONS

- First, the try block is executed. If there are no errors, except is skipped.
- If there are errors, the rest of the try block is skipped.
  - Proceeds to except block with the matching exception type.
- Execution proceeds as normal.

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...
Enter a number: two
Ooops !! That was not a valid number. Try again.
Enter a number: 100
```

66

# HANDLING EXCEPTIONS

The try/except clause options are as follows:

| Clause form | Interpretation |
|---|---|
| `except:` | Catch all (or all other) exception types |
| `except name:` | Catch a specific exception only |
| `except name as value:` | Catch the listed exception and its instance |
| `except (name1, name2):` | Catch any of the listed exceptions |
| `except (name1, name2) as value:` | Catch any of the listed exceptions and its instance |
| `else:` | Run if no exception is raised |
| `finally:` | Always perform this block |



# HANDLING EXCEPTIONS

There are a number of ways to form a try/except block.

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...     except (TypeError, IOError) as e:
...         print(e)
...     else:
...         print("No errors encountered!")
...     finally:
...         print("We may or may not have encountered errors…")
...
```

# RAISING AN EXCEPTION

Use the raise statement to force an exception to occur. Useful for diverting a program or for raising custom exceptions.

```
try:
    raise IndexError("Index out of range")
except IndexError as ie:
    print("Index Error occurred: ", ie)
```

Output:
Index Error occurred:  Index out of range

69

# STRINGS

We've already introduced the string data type .

Strings are sequences.

Strings are written with either single or double quotes encasing a sequence of characters.

```
s1 = ' STAY AT HOME! '

s2 = "It is fun to learn Python."
```

**Note that there is no character data type in Python**. A character is simply represented as a string with one character.

70

# ACCESSING STRINGS

As a sequence data type, strings can be accessed element-wise as they are technically just sequences of character elements.

We can index with typical bracket notation, as well as perform slicing.

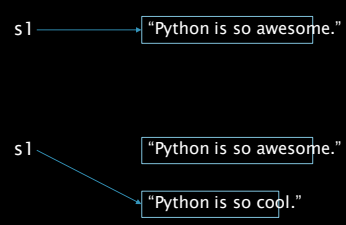
```
>>> s1 = ”STAY AT HOME!"
>>> s2 = "Python is so awesome."
>>> print s1[3]
Y
>>> print s2[5:15]
n is so aw
```

71

# MODIFYING STRINGS

Strings are *immutable* – you cannot update the value of an existing string object. However, you can reassign your variable name to a new string object to perform an “update”.

```
>>> s1 = "Python is so awesome."
>>> s1 = "Python is so cool."
```




72

# MODIFYING STRINGS

Alternatively, we could have done the following:

```
>>> s1 = "Python is so awesome."
>>> s1 = s1[:13] + "cool."
```

This will create a substring "Python is so ", which is concatenated with "cool.", stored in memory and associated with the name s1.

The **"+"** operator can be used with two string objects to **concatenate** them together. The **"*"** operator can be used to **concatenate multiple copies of a single string object**.

We also have **`in`** and **`not in`** available for testing character membership within a string.

73

# ESCAPE CHARACTERS

As a side note, there are a number of escape characters supported by Python strings. The most common ones are:

- '\n' – newline
- '\s' – space
- '\t' – tab

74

# BUILT-IN STRING METHODS

Python includes a number of built-in string methods that are incredibly useful for string manipulation. Note that these *return* the modified string value; we cannot change the string's value in place because they're immutable!

- **`s.upper()`** and **`s.lower()`**

```
>>> s1 = "Python is so awesome."
>>> print s1.upper()
PYTHON IS SO AWESOME.
>>> print s1.lower()
python is so awesome.
```

75

# BUILT-IN STRING METHODS

- **`s.isalpha()`**, **`s.isdigit()`**, **`s.isalnum()`**, **`s.isspace()`** – return True if string *s* is composed of alphabetic characters, digits, either alphabetic and/or digits, and entirely whitespace characters, respectively.
- **`s.islower()`**, **`s.isupper()`** – return True if string *s* is all lowercase and all uppercase, respectively.

```
>>> "WHOA".isupper()
True
>>> "12345".isdigit()
True
>>> " \n ".isspace()
True
>>> "hello!".isalpha()
False
```

76

# BUILT-IN STRING METHODS

- **`str.split([sep[, maxsplit]])`** – Split *str* into a list of substrings. The *sep* argument indicates the delimiting string (defaults to consecutive whitespace). The *maxsplit* argument indicates the maximum number of splits to be done (default is -1).
- `str.rsplit([sep[, maxsplit]])` – Split *str* into a list of substrings, starting from the right.
- `str.strip([chars])` – Return a copy of the string *str* with leading and trailing characters removed. The *chars* string specifies the set of characters to remove (default is whitespace).
- `str.rstrip([chars])` – Return a copy of the string *str* with only trailing characters removed.

77

# BUILT-IN STRING METHODS

```
>>> "Python programming is fun!".split()
['Python', 'programming', 'is', 'fun!']
>>> "555-867-5309".split('-')
['555', '867', '5309']
>>> "***Python programming is fun***".strip('*')
'Python programming is fun'
```

78

# BUILT-IN STRING METHODS

- ***str*.find(*sub*[, *start*[, *end*]])** – return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the slice *str*[*start*:*end*]. Return -1 if *sub* is not found. See also `str.rfind()`.
- `str.index(sub[, start[, end]])` – identical to find(), but raises a *ValueError* exception when substring *sub* is not found. See also `str.rindex()`.
- `str.join(iterable)` – return a string that is the result of concatenating all of the elements of *iterable*. The *str* object here is the delimiter between the concatenated elements.
- ***str*.replace(*old*, *new*[, *count*])** – return a copy of the string *str* where all instances of the substring *old* are replaced by the string *new* (up to *count* number of times).

79

# BUILT-IN STRING METHODS

```
>>> "whenever".find("never")
3
>>> "whenever".find("what")
-1
>>> "whenever".index("what")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> "-".join(['555','867','5309'])
'555-867-5309'
>>> " ".join(['Python', 'is', 'awesome'])
'Python is awesome'
>>> "whenever".replace("ever", "ce")
'whence'
```

80

# THE STRING MODULE

Additional built-in string methods may be found here.

All of these built-in string methods are methods of any string object. They do not require importing any module or anything - they are part of the core of the language.

There is a string module, however, which provides some additional useful string tools. It defines useful string constants, the string formatting class, and some deprecated string functions which have mostly been converted to methods of string objects.

81

# STRING CONSTANTS

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

82

# STRING CONSTANTS

```
>>> import string
>>> string.lowercase #locale-dependent
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase #locale-dependent
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.letters # lowercase+uppercase
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.octdigits
'01234567'
>>> print string.punctuation
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

83