كلية الحاسبات والمعلومات

كلية معتمدة من الهيئة القومية لضمان جودة التعليم والاعتماد

Assiut University

Course Title: Data Structures and Algorithms
Course Code: CS211

Prof. Dr. Khaled F. Hussain

Reference

- Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, “Data Structures and Algorithms in Python”

Objects in Python

- Python is an object-oriented language, and ***classes*** form the basis for all data types.
- Identifiers in Python are ***case-sensitive.***
- Unlike Java and C++, Python is a ***dynamically typed*** language, as there is no advance declaration associating an identifier with a particular data type. An identifier can be associated with any type of object, and it can later be reassigned to another object of the same (or different) type.
- A programmer can establish an ***alias*** by assigning a second identifier to an existing object.
  - Once an alias has been established, either name can be used to access the underlying object.
  - However, if one of the *names* is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias.
  - Example:
    - t1 = 50.5
    - t2=t1 # t1 and t2 points to the float object contains 50.5
    - t1=t1+10 # The result is stored as a new floating-point instance.
    - # t1 points to the float object contains 60.5 and t2 points to the float object contains 50.5

Creating and Using Objects

- The process of creating a new instance of a class is known as ***instantiation***.
  - In general, the syntax for instantiating an object is to invoke the ***constructor*** of a class.
- A class is ***immutable*** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed. For example, the float class is immutable.
  - bool, int, float, tuple, str, frozenset classes are ***immutable***
  - list, set, dict classes are **not** ***immutable***

Creating and Using Objects (Cont.)

- There are four collection data types in the Python programming language:
  - **List** is a collection which is ordered and changeable (***not immutable)*. Allows duplicate members.
  - **Tuple** is a collection which is ordered and unchangeable (***immutable*). Allows duplicate members.
  - **Set** is a collection which is unordered, changeable, and unindexed. No duplicate members.
  - **Dictionary** is a collection which is ordered and changeable. No duplicate members.

Creating and Using Objects (Cont.)

- The bool Class*:*
    - The default constructor, bool( ), returns False
    - Python allows the creation of a Boolean value from a nonboolean type using the syntax bool(foo) for value foo.
    - Numbers evaluate to False if zero, and True if nonzero.
    - strings and lists, evaluate to False if empty and True if nonempty.

Creating and Using Objects (Cont.)

- The int Class
  - Unlike Java and C++, which support different integral types with different precisions (e.g., `int`, `short`, `long`), Python automatically chooses the internal representation for an integer based upon the magnitude of its value.
  - Example of such literals are respectively `0b1011`, `0o52`, and `0x7f` (binary, octal, hexadecimal).
  - The integer constructor, `int( )`, returns value 0 by default.
  - For example, if `f` represents a floating-point value, the syntax `int(f)` produces the *truncated* value of `f`. For example, both `int(3.14)` and `int(3.99)` produce the value 3

Creating and Using Objects (Cont.)

- The str Class
  - str class is specifically designed to efficiently represent an immutable sequence of characters, based upon the Unicode international character set.
  - String literals can be enclosed in single quotes, as in 'hello' , or double quotes, as in "hello".
  - Python also supports using the delimiter or """ to begin and end a string literal. The advantage of such triple-quoted strings is that newline characters can be embedded naturally (rather than escaped as \n).
  - Unicode characters can be included, such as '20\u20AC' for the string 20€.

Creating and Using Objects (Cont.)

- The tuple Class
    - The tuple class provides an immutable version of a sequence
    - () being an empty tuple.
    - To express a tuple of length one as a literal, a comma must be placed after the element
    - For example, (17,) is a one-element tuple- the expression (17) is viewed as a simple parenthesized numeric expression.

The set and frozenset Classes

- The set and frozenset Classes
  - Python's set class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements.
  - This is based on a data structure known as a ***hash table***.
  - The set does not maintain the elements in any particular order.
  - Only instances of ***immutable*** types can be added to a Python set such as integers, floating-point numbers, and character strings
  - The frozenset class is an immutable form of the set type, so it is legal to have a set of frozensets.
  - Python uses curly braces { and } as delimiters for a set.
  - {17} or { red , green , blue }.
  - The exception to this rule is that { } does not represent an empty set; for historical reasons, it represents an empty dictionary
  - Instead, the constructor syntax set( ) produces an empty set.
  - For example, set('hello') produces { 'h' , 'e' , 'l' , 'o' }.

The set and frozenset Classes (Cont.)

- Sets and frozensets support the following operators:

key in s	containment check
key not in s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 | s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2
s1 − s2	the set of elements in s1 but not s2
s1 ˆ s2	the set of elements in precisely one of s1 or s2

The set and frozenset Classes (Cont.)

```
myset = {"apple", "banana", "cherry"}
print(myset)
```

- Note: The values True and 1 are considered the same value in sets, and are treated as duplicates:
    - False and 0 is considered the same value:

```
thisset = {"apple", "banana", "cherry", True, 1, 2}
print(thisset)
```

The set and frozenset Classes (Cont.)

```
print(len(thisset))
```

- Using the set() constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)

thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

The set and frozenset Classes (Cont.)

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

- To add items from another set into the current set, use the update() method.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```

The set and frozenset Classes (Cont.)

- The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)

thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

The set and frozenset Classes (Cont.)

- Remove a random item by using the pop() method:

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)

thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

The set and frozenset Classes (Cont.)

- The del keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset) # Error
```

The set and frozenset Classes (Cont.)

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

- The union() method allows you to join a set with other data types, like lists or tuples.
- Join a set with a tuple:

```
x = {"a", "b", "c"}
y = (1, 2, 3)
z = x.union(y)
print(z)
```

The set and frozenset Classes (Cont.)

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1.intersection(set2)
print(set3)
```

- Use & for intersection of sets:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 & set2
print(set3)
```

- The & operator only allows you to join sets with sets, and not with other data types like you can with the intersection() method.

The set and frozenset Classes (Cont.)

```
animals = frozenset(["cat", "dog", "lion"])
print("cat" in animals)
print("elephant" in animals)

animals = ["cat", "dog", "lion"]
 # converting list to frozenset
animals2 = frozenset(animals)
print("frozenset Object is : ", animals2)
```

The set and frozenset Classes (Cont.)

```
# initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])

# copying a frozenset
C = A.copy()
print(C)

# union
union_set = A.union(B)
print(union_set)
```

The set and frozenset Classes (Cont.)

```
# intersection
intersection_set = A.intersection(B)
print(intersection_set)

difference_set = A.difference(B)
print(difference_set)

# symmetric_difference
symmetric_difference_set = A.symmetric_difference(B)
print(symmetric_difference_set)
```

- The ^ operator only allows you to join sets with sets, and not with other data types like you can with the symmetric_difference() method.

The set and frozenset Classes (Cont.)

```
Z_union=A | B
print(Z_union)

Z_intersection =A & B
print(Z_intersection)
```

The dict Class

- The dict Class
  - Python's dict class represents a ***dictionary***, or ***mapping***, from a set of distinct ***keys*** to associated ***values***.
  - For example, a dictionary might map from unique student ID numbers, to larger student records (such as the student's name, address, and course grades).
  - Python implements a dict using an almost identical approach to that of a set, but with storage of the associated values.
  - A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form { } produces an empty dictionary.
  - For example, the dictionary thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964 }

The dict Class (Cont.)

- d[key] value associated with given key
- d[key] = value set (or reset) the value associated with given key
- del d[key] remove key and its associated value from dictionary
- key in d containment check
- key not in d non-containment check
- d1 == d2 d1 is equivalent to d2
- d1 != d2 d1 is not equivalent to d2

The dict Class (Cont.)

```
thisdict={"brand":"Ford","model":"Mustang","year":19
64}
print(len(thisdict))
```

- Get the value of the "model" key:

```
print(thisdict["model"])
```

- or

```
print(thisdict.get("model"))
```

The dict Class (Cont.)

- Get Keys

```
print(thisdict.keys())
```

- Add a new item to the original dictionary

```
thisdict["color"]="white"
print(thisdict)
```

- Get a list of the values:
- print(thisdict.values())

The dict Class (Cont.)

- Make a change in the dictionary

```
thisdict["year"]=2020
```

- Or

```
thisdict.update({"year":2020})
```

- Get each item in a dictionary, as tuples in a list.

```
print(thisdict.items())
```

The dict Class (Cont.)

- Using the dict() method to make a dictionary:

```
thisdict=dict(name="John", age =
36,country="Norway")
print(thisdict)
```

The dict Class (Cont.)

- Dictionaries do not save two items with the same key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

- Result:
- {'brand': 'Ford', 'model': 'Mustang', 'year': 2020}

The dict Class (Cont.)

- Check if "model" is present in the dictionary:

```
if "model" in thisdict:
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

- The pop() method removes the item with the specified key name:

```
thisdict.pop("model")
print(thisdict)
```

- The popitem() method removes the last inserted item:

```
thisdict.popitem()
print(thisdict)
```

The dict Class (Cont.)

- The del keyword removes the item with the specified key name:

```
del thisdict["model"]
print(thisdict)
```

- The del keyword can also delete the dictionary completely:

```
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

- The clear() method empties the dictionary:

```
thisdict.clear()
print(thisdict)
```

The dict Class (Cont.)

- Print all key names in the dictionary:

```
for x in thisdict:
    print(x)
```

- Print all *values* in the dictionary

```
for x in thisdict:
    print(thisdict[x])
```

- Or

```
for x in thisdict.values():
    print(x)
```

The dict Class (Cont.)

- Loop through both keys and values, by using the items() method:

```
for x, y in thisdict.items():
  print(x, y)
```

The list Class

- The list Class
  - The list class is the most general, representing a sequence of arbitrary objects (akin to an “array” in other languages).
  - A list is a ***referential*** structure, as it technically stores a sequence of ***references*** to its elements.
  - Lists are ***array-based*** sequences and are ***zero-indexed***
  - The list( ) constructor produces an empty list by default.
  - [] is an empty list.
  - ["red","green","blue"] is a list containing three string instances.
  - The syntax list(“hello”) produces a list of individual characters, [‘h’ , ‘e’ , ‘l’ , ‘l’ , ‘o’].
  - Because lists are mutable, the syntax s[j] = val can be used to replace an element at a given index. Lists also support a syntax, del s[j], that removes the designated element from the list.

The list Class (Cont.)

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
print(len(thislist))
print(thislist[1])
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
list1 = ["abc", 34, True, 40, "male"]
```

The list Class (Cont.)

- -1 refers to the last item, -2 refers to the second last item etc.

```
print(thislist[-1])
```

- Using the list() constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

Logical Operators

- not (unary negation)
- and (conditional and)
- or (conditional or)

Equality Operators

- `is` (same identity; a is b, `a` and `b` are aliases for the *same* object.)
- `is not` (different identity)
- `==` (equivalent; If identifiers `a` and `b` refer to the same object, then `a == b` should also evaluate to `True`. Yet `a == b` also evaluates to `True` when the identifiers refer to different objects that happen to have same values)
- `!=` (not equivalent)

- In most programming situations, the equivalence tests `==` and `!=` are the appropriate operators; use of `is` and `is not` should be reserved for situations in which it is necessary to detect true aliasing.

Equality Operators (Cont.)

```
x = [1, 2, 3]
y = [1, 2, 3]
z = x
if x == y:
   print("True")
else:
   print("False")
```

Result:

True

Equality Operators (Cont.)

```
x = [1, 2, 3]
y = [1, 2, 3]
z = x
# Case 1: Identity comparison (is)
if x is y:
   print("True")
else:
   print("False")

# Case 2: Comparing references (is)
if x is z:
   print("True")
else:
   print("False")
```

Result:

False

True

Arithmetic Operators

- `/` (true division)
- `//` (integer division)

- In Python, the `/` operator designates ***true division***, returning the floating-point result of the computation.
- Thus, `27 / 4` results in the `float` value `6.75`.
- The expression `27 // 4` evaluates to `int` value 6 (the mathematical *floor* of the quotient).

Bitwise Operators

- ~ bitwise complement (prefix unary operator)
- & bitwise and
- | bitwise or
- ^ bitwise exclusive-or
- << shift bits left, filling in with zeros
- >> shift bits right, filling in with sign bit

```
a = 10
b = 4
# Print bitwise AND operation
print("a & b =", a & b)
```

- Resut:

a & b = 0

Sequence Operators

- s[j] element at index *j*
- s[start:stop] slice including indices [start,stop). For example, the syntax data[3:8] denotes a subsequence including the five indices: 3,4,5,6, 7.
- s[start:stop:step] slice including indices start, start + step, start + 2 step, . . . , up to but not equalling or stop
  - If a start index or stop index is omitted in the slicing notation, it is presumed to designate the respective extreme of the original sequence.
- s + t concatenation of sequences
- K * s shorthand for s + s + s + ... (k times)

```
result = "Hello, world!" * 5
print(result)
```

- val in s containment check. For strings, this syntax can be used to check for a single character or for a larger substring.

```
print("amp" in "example")
```

- val not in s non-containment check

Sequence Operators (Cont.)

```
my_list = ['a', 'b', 'c', 'd','e']
print(my_list[2])

#slicing (start, stop)
print(my_list[1:4])

#(start,stop,step)
#Slices the sequence from index i to j with a step k.
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers[1:8:2])
```

- Results:

```
c
['b', 'c', 'd']
[1, 3, 5, 7]
```

Control Flow

- A body is more typically typeset as an ***indented block*** starting on the line following the colon. Python relies on the indentation level to designate the extent of that block of code

```
if door is closed:
    if door is locked:
        unlock door( )
    open door( )
advance()
```

Control Flow (Cont.)

```
j = 0
while j < len(data) and data[j] != X :
    j += 1

total = 0
for val in data:
    total += val

biggest = data[0]
for val in data:
    if val > biggest:
        biggest = val
```

Control Flow (Cont.)

- range(n) generates the series of n values from 0 to $n-1$.

```
big_index = 0
for j in range(len(data)):
    if data[j] > data[big _index]:
        big_index = j

found = False
for item in data:
    if item == target:
        found = True
        break
```

Functions

```
def count(data, target):
    n = 0
    for item in data:
        if item == target:          # found a match
            n += 1
    return n
```

- Unlike Java and C++, Python is a dynamically typed language, and therefore a Python Function signature does not designate the types of those parameters, nor the type (if any) of a return value.
- Parameter passing in Python follows the semantics of the standard ***assignment statement***.
    - prizes = count(grades, ‘A’)
        - data = grades
        - target = ‘A’

Functions (Cont.)

- These assignment statements establish identifier data as an alias for grades and target.
- The communication of a return value from the function back to the caller is similarly implemented as an assignment.
- An advantage to Python's mechanism for passing information to and from a function is that objects are not copied. This ensures that the invocation of a function is efficient, even in a case where a parameter or return value is a complex object.
- Mutable Parameters: we note that reassigning a new value to a formal parameter with a function body, such as by setting data = [ ], does not alter the actual parameter; such a reassignment simply breaks the alias.
- Default Parameter Values:
  - def foo(a, b=15, c=27):
  - Python supports an alternate mechanism for sending a parameter to a function known as a ***keyword argument***.
  - For example, with signature foo(a=10, b=20, c=30) call foo(c=5) will invoke the function with parameters a=10, b=20, c=5.

Functions (Cont.)

- Keyword Arguments

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "c1", child2 = "C2", child3 = "C3")
```

Functions (Cont.)

```
def my_function(country = "Egypt"):
  print("I am from " + country)

my_function("Sweden")
my_function()
```

Functions (Cont.)

```
def compute_gpa(grades, points={ "A+" :4.0, "A" :4.0, "A-" :3.67, "B+" :3.33, "B" :3.0, "B-" :2.67, "C+" :2.33, "C"
:2.0, "C-" :1.67, "D+" :1.33, "D" :1.0, "F":0.0}):
    num_courses = 0
    total_points = 0
    for g in grades:
        if g in points: # a recognizable grade
            num_courses += 1
            total_points += points[g]
    return total_points / num_courses
```

Python’s Built-In Function

- Input/Output: print, input(prompt) (Return a string from standard input; the prompt is optional), and open(filename, mode) (Open a file with the given name and access mode).
- Character Encoding:ord and chr; For example, ord(‘A’) is 65 and chr(65) is ‘A’
- Mathematics: abs, pow, round, sum, and divmod(x, y) (Return (x // y, x % y) as tuple, if x and y are integers).
- Ordering: max, min, and sorted(iterable) (Return a list containing elements of the iterable in sorted order)
- Collections/Iterations: range generates a new sequence of numbers; len reports the length of any existing collection; iter and next provide a general framework for iteration through elements of a collection.

Python’s Built-In Function (Cont.)

year = int(input(“In what year were you born?”))

reply = input(“Enter x and y, separated by spaces: “)

pieces = reply.split( ) # returns a list of strings, as separated by spaces

x = float(pieces[0])

y = float(pieces[1]

Files

- fp = open(“sample.txt”), attempts to open a file named sample.txt, returning a proxy that allows read-only access to the text file.
- fp = open(“results.txt” , ‘w’)
- The syntax fp.close( ) closes the file associated with proxy fp, ensuring that any written contents are saved.

Files (Cont.)

- fp.read( ) Return the (remaining) contents of a readable file as a string.
- fp.read(k) Return the next k bytes of a readable file as a string.
- fp.readline( ) Return (remainder of) the current line of a readable file as a string.
- fp.readlines( ) Return all (remaining) lines of a readable file as a list of strings. for line in fp: Iterate all (remaining) lines of a readable file.
- fp.seek(k) Change the current position to be at the kth byte of the file.
- fp.tell( ) Return the current position, measured as byte-offset from the start.
- fp.write(string) Write given string at current position of the writable file.
- fp.writelines(seq) Write each of the strings of the given sequence at the current position of the writable file. This command does not insert any newlines, beyond those that are embedded in the strings.
- print(..., file=fp) Redirect output of print function to the file

Raising an Exception

```
raise ValueError("x cannot be negative" )


def sqrt(x):
    if not isinstance(x, (int, float)):
        raise TypeError("x must be numeric" )
    elif x < 0:
        raise ValueError("x cannot be negative" )
```

Catching an Exception using a **try-except**

```
try:
   ratio = x / y
except ZeroDivisionError:
   … do something else ..
```

```
try:
   fp = open( “sample.txt” )
except IOError as e:
   print( “Unable to open the file:” , e)
```

Python Iterators

- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- In Python, an iterator is an object which impanements the iterator protocol, which consist of the methods __iter__() and __next__().
- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.
  - All these objects have an iter() method which is used to get an iterator.
- strings are iterable objects, and can return an iterator.

Python Iterators (Cont.)

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

Python Iterators (Cont.)

```
mystr = "banana"
myit = iter(mystr)
print(next(myit))
print(next(myit))
print(next(myit))
```

- Or

```
mystr = "banana"
for x in mystr:
  print(x)
```

Python Iterators (Cont.)

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)
```

Python Iterators (Cont.)

- Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
```

Python Iterators (Cont.)

- An ***iterator*** for a collection provides one key behavior: It supports a special method named next that returns the next element of the collection, if any, or raises a StopIteration exception to indicate that there are no further elements.

```
class SequenceIterator:
  def __init__(self, sequence):
    self._seq = sequence # keep a reference to the underlying data
    self._k = −1 # will increment to 0 on first call to next

  def __next__(self):
    self._k += 1 # advance to next index
    if self._k < len(self._seq):
      return(self._seq[self._k]) # return the data element
  else:
    raise StopIteration( ) # there are no more elements

  def __iter__(self):
    return self
```

Python Iterators (Cont.)

- Python also supports functions and classes that produce an implicit iterable series of values, that is, without constructing a data structure to store all of its values at once.
- For example, the call `range(1000000)` does *not* return a list of numbers; it returns a `range` object that is iterable. This object generates the million values one at a time, and only as needed.
- Such a ***lazy evaluation*** technique has great advantage. In the case of range, it allows a loop of the form, `for j in range(1000000):`, to execute without setting aside memory for storing one million values.

Generator

- However, the most convenient technique for creating iterators in Python is through the use of ***generators***.
- A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement.
- As an example, consider the goal of determining all factors of a positive integer. For example, the number 100 has factors 1, 2, 4, 5, 10, 20, 25, 50, 100.
- A traditional function might produce and return a list containing all factors, implemented as:

```
def factors(n): # traditional function that computes factors
    results = [ ] # store factors in a new list
    for k in range(1,n+1):
        if n % k == 0: # divides evenly, thus k is a factor
            results.append(k) # add k to the list of factors
    return results # return the entire list
```

Generator (Cont.)

- Notice use of the keyword yield rather than return to indicate a result. This indicates to Python that we are defining a generator, rather than a traditional function.

```
def factors(n): # generator that computes factors
    for k in range(1,n+1):
        if n % k == 0: # divides evenly, thus k is a factor
            yield k # yield this factor as next result

for x in factors(10):
    print(x)
```

Conditional Expressions

- This compound expression evaluates to *expr1* if the condition is true, and otherwise evaluates to *expr2*.

```
expr1 if condition else expr2
```

```
if n >= 0:
    param = n
else:
    param = −n
```

```
param = n if n >= 0 else −n
```

```
result = foo(n if n >= 0 else −n)
```

Comprehension Syntax

```
[ expression for value in iterable if condition ]
```

- The evaluation of the comprehension is logically equivalent to the following traditional control structure

```
result = [ ]
for value in iterable:
    if condition:
        result.append(expression)
```

```
squares = [k*k for k in range(1, n+1)]
```

- The evaluation of the comprehension is logically equivalent to the following traditional control structure

```
squares = [ ]
for k in range(1, n+1):
    squares.append(k*k)
```

Comprehension Syntax (Cont.)

Python supports similar comprehension syntaxes that respectively produce a set, generator, or dictionary. We compare those syntaxes using our example for producing the squares of numbers.

[k*k for k in range(1, n+1)] list comprehension

{ k*k for k in range(1, n+1) } set comprehension

(k*k for k in range(1, n+1)) generator comprehension

{ k : k*k for k in range(1, n+1) } dictionary comprehension

Packing and Unpacking of Sequences

- If a series of comma-separated expressions are given in a larger context, they will be treated as a single tuple, even if no enclosing parentheses are provided.
- For example, the assignment `data = 2, 4, 6, 8` results in identifier, `data`, being assigned to the tuple `(2, 4, 6, 8)`.
- This behavior is called ***automatic packing*** of a tuple.
- If the body of a function executes the command, `return x, y`. it will be formally returning a single object that is the tuple `(x, y)`.
- Python can automatically ***unpack*** a sequence, allowing one to assign a series of individual identifiers to the elements of sequence. As an example, we can write

  ```
  a, b, c, d = range(7, 11)
  ```

  which has the effect of assigning `a=7`, `b=8`, `c=9`, and `d=10`
- This technique can be used to unpack tuples returned by a function.

  ```
  quotient, remainder = divmod(a, b)
  ```

  ```
  for x, y in [ (7, 2), (5, 8), (6, 4) ]:
  ```

Simultaneous Assignments

- The combination of automatic packing and unpacking forms a technique known as ***simultaneous assignment***

```
x, y, z = 6, 2, 5
```

First-Class Objects

- ***first-class objects*** are instances of a type that can be assigned to an identifier, passed as a parameter, or returned by a function.
- For example, we could write the following:

```
scream = print          # assign name 'scream' to the function denoted as 'print'
scream("Hello" )        # call that function
```

- In this case, we have not created a new function, we have simply defined `scream` as an alias for the existing `print` function.
- This mechanism is used by Python to allow one function to be passed as a parameter to another.

Modules and the Import Statement

- Beyond the built-in definitions, the standard Python distribution includes perhaps tens of thousands of other values, functions, and classes that are organized in additional libraries, known as ***modules***, that can be ***imported*** from within a program.
- As an example, we consider the `math` module. While the built-in namespace includes a few mathematical functions (e.g., `abs`, `min`, `max`, `round`), many more are relegated to the `math` module (e.g., `sin`, `cos`, `sqrt`). That module also defines approximate values for the mathematical constants, `pi` and `e`.
- Python's `import` statement loads definitions from a module into the current namespace.

```
from math import pi, sqrt
```

- This command adds both `pi` and `sqrt`, as defined in the `math` module, into the current namespace, allowing direct use of the identifier, `pi`, or a call of the function, `sqrt(2)`.
- If there are many definitions from the same module to be imported, an asterisk may be used as a wild card, as in, `from math import *`
  - The danger of that is the names defined in the module may conflict with names already in the current namespace (or being imported from another module), and the import causes the new definitions to replace existing ones.

Modules and the Import Statement (Cont.)

- Another approach that can be used to access many definitions from the same module is to import the module itself, using a syntax such as:

```
import math
```

- Formally, this adds the identifier, `math`, to the current namespace, with the module as its value.
- Once imported, individual definitions from the module can be accessed using a fully-qualified name, such as `math.pi` or `math.sqrt(2)`.

Creating a New Module

- To create a new module, one simply has to put the relevant definitions in a file named with a `.py` suffix.
- For example, if we were to put the definition of a `count` function into a file named `utility.py`, we could import that function using the syntax, `from utility import count`.

Existing Modules

- Summary of a few available modules that are relevant to a study of data structures.
- array — Provides compact array storage for primitive types.
- collections — Defines additional data structures and abstract base classes involving collections of objects.
- copy — Defines general functions for making copies of objects.
- heapq — Provides heap-based priority queue functions.
- math — Defines common mathematical constants and functions.
- os — Provides support for interactions with the operating system.
- random — Provides random number generation.
- re — Provides support for processing regular expressions.
- sys — Provides additional level of interaction with the Python interpreter.
- time — Provides support for measuring time, or delaying a program.

Pseudo-Random Number Generation

- seed(hashable) Initializes the pseudo-random number generator based upon the hash value of the parameter
- random() Returns a pseudo-random floating-point value in the interval [0.0,1.0).
- randint(a,b) Returns a pseudo-random integer in the closed interval [*a*,*b*].
- randrange(start, stop, step) Returns a pseudo-random integer in the standard Python range indicated by the parameters.
- choice(seq) Returns an element of the given sequence chosen pseudo-randomly.
- shuffle(seq) Reorders the elements of the given sequence pseudo-randomly.

- Return a random element from a list:

```
import random

mylist = ["apple", "banana", "cherry"]

print(random.choice(mylist))
```