**ADVANCED TOPICS IN PYTHON**

**Iterators for Dictionaries**

Let's start with iterating over a dictionary. Recall that a dictionary is just a collection of keys and values.

d = { "Name": "Guido", "Age": 56, "BDFL": True } print d.items() # => [('BDFL', True), ('Age', 56), ('Name', 'Guido')]

Note that the .items() method doesn't return key/value pairs in any specific order.

**1.**

Create your own Python dictionary, my\_dict, in the editor to the right with two or three key/value pairs.

Then, print the result of calling the my\_dict.items().

Hint

You can think of dictionaries as unordered key/value pairs.

|  |
| --- |
| my\_dict = {  "Name": "Lucy",  "Age": 56,  "BDFL": True  }  print my\_dict.items() |

**keys() and values()**

While .items() returns an array of *tuples*with each tuple consisting of a key/value pair from the dictionary:

* The .keys() method returns a list of the dictionary's keys, and
* The .values() method returns a list of the dictionary's values.

Again, these methods will not return the keys or values from the dictionary in any specific order.

You can think of a tuple as an immutable (that is, unchangeable) list. Tuples are surrounded by ()s and can contain any data type.

**Instructions**

**1.**

Remove your call to .items() and replace it with a call to .keys() and a call to .values(), each on its own line. Make sure to print both!

Hint

For instance, your call to my\_dict.keys() might look like:

print my\_dict.keys()

|  |
| --- |
| print my\_dict.keys()  print my\_dict.values() |

**The 'in' Operator**

For iterating over lists, tuples, dictionaries, and strings, Python also includes a special keyword: in. You can use in very intuitively, like so:

for number in range(5): print number, d = { "name": "Eric", "age": 26 } for key in d: print key, d[key], for letter in "Eric": print letter, # note the comma!

1. In the example above, first we create and iterate through a range, printing out 0 1 2 3 4. Note that the trailing comma ensures that we keep printing on the same line.
2. Next, we create a dictionary and iterate through, printing out age 26 name Eric. Dictionaries have no specific order.
3. Finally, we iterate through the letters of a string, printing out E r i c.

**1.**

For each key in my\_dict: print out the key , then a space, then the value stored by that key. (You should use print a, b rather than print a + " " + b.)

Hint

You'll want to do something like this:

for key in some\_dict: print key, some\_dict[key]

|  |
| --- |
| for key in my\_dict:  print key , " " , my\_dict[key] |

# Building Lists

Let's say you wanted to build a list of the numbers from 0 to 50 (inclusive). We could do this pretty easily:

my\_list = range(51)

But what if we wanted to generate a list according to some logic—for example, a list of all the even numbers from 0 to 50?

Python's answer to this is the **list comprehension**. List comprehensions are a powerful way to generate lists using the for/in and if keywords we've learned.

**Instructions**

**1.**

Check out the list comprehension example in the editor. When you're pretty sure you know what it'll do, click Run to see it in action.

|  |
| --- |
| evens\_to\_50 = [i for i in range(51) if i % 2 == 0]  print evens\_to\_50 |

# List Comprehension Syntax

Here's a simple example of list comprehension syntax:

new\_list = [x for x in range(1, 6)] # => [1, 2, 3, 4, 5]

This will create a new\_list populated by the numbers one to five. If you want those numbers doubled, you could use:

doubles = [x \* 2 for x in range(1, 6)] # => [2, 4, 6, 8, 10]

And if you only wanted the doubled numbers that are evenly divisible by three:

doubles\_by\_3 = [x \* 2 for x in range(1, 6) if (x \* 2) % 3 == 0] # => [6]

**Instructions**

**1.**

Use a list comprehension to build a list called even\_squares in the editor.

Your even\_squares list should include the squares of the even numbers between 1 to 11. Your list should start [4, 16, 36...] and go from there.

Hint

You can use x \*\* 2 to square a number, and x % 2 == 0 to check if it's even.

|  |
| --- |
| doubles\_by\_3 = [x \* 2 for x in range(1, 6) if (x \* 2) % 3 == 0]  # Complete the following line. Use the line above for help.  even\_squares = [x \*\* 2 for x in range(1, 12) if x%2 == 0]  print even\_squares |

# Now You Try!

Great work! Now it's time for you to create a list comprehension all on your own.

c = ['C' for x in range(5) if x < 3] print c

The example above creates and prints out a list containing ['C', 'C', 'C'].

**Instructions**

**1.**

Use a list comprehension to create a list, cubes\_by\_four.

The comprehension should consist of the cubes of the numbers 1 through 10 only if the cube is evenly divisible by four.

Finally, print that list to the console.

Note that in this case, the cubed number should be evenly divisible by 4, not the original number.

|  |
| --- |
| cubes\_by\_four = [x\*\*3 for x in range(1,11) if (x\*\*3)%4==0]  print cubes\_by\_four |

# List Slicing Syntax

Sometimes we only want part of a Python list. Maybe we only want the first few elements; maybe we only want the last few. Maybe we want every other element!

List slicing allows us to access elements of a list in a concise manner. The syntax looks like this:

[start:end:stride]

Where start describes where the slice starts (inclusive), end is where it ends (exclusive), and stride describes the space between items in the sliced list. For example, a stride of 2 would select every other item from the original list to place in the sliced list.

|  |
| --- |
| l = [i \*\* 2 for i in range(1, 11)]  # Should be [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  print l[2:9:2] |
| [9, 25, 49, 81] |

**Omitting Indices**

If you don't pass a particular index to the list slice, Python will pick a default.

to\_five = ['A', 'B', 'C', 'D', 'E'] print to\_five[3:] # prints ['D', 'E'] print to\_five[:2] # prints ['A', 'B'] print to\_five[::2] # print ['A', 'C', 'E']

1. The default starting index is 0.
2. The default ending index is the end of the list.
3. The default stride is 1.

**Instructions**

**1.**

Use list slicing to print out every odd element of my\_list from start to finish.

Omit the start and end index. You only need to specify a stride.

Check the Hint if you need help.

Hint

Remember, the syntax for list slicing is

[start:end:stride]

Since you're using the entire list, you should leave out the start and end indices (but leave in the colons!) and give the slice a stridethat will select every other (that is, odd) element.

|  |
| --- |
| my\_list = range(1, 11) # List of numbers 1 - 10  # Add your code below!  print my\_list[::2] |

# Reversing a List

We have seen that a positive stride progresses through the list from left to right.

A negative stride progresses through the list from right to left.

letters = ['A', 'B', 'C', 'D', 'E'] print letters[::-1]

In the example above, we print out ['E', 'D', 'C', 'B', 'A'].

**Instructions**

**1.**

Create a variable called backwardsand set it equal to the reversed version of my\_list.

|  |
| --- |
| my\_list = range(1, 11)  backwards = my\_list[::-1] |

# Stride Length

A positive stride length traverses the list from left to right, and a negative one traverses the list from right to left.

Further, a stride length of 1 traverses the list "by ones," a stride length of 2 traverses the list "by twos," and so on.

**1.**

Create a variable, backwards\_by\_tens, and set it equal to the result of going backwards through to\_one\_hundredby tens. Go ahead and print backwards\_by\_tens to the console.

Hint

Remember, the syntax is:

new\_list = old\_list[begin:end:stride]

|  |
| --- |
| to\_one\_hundred = range(101)  backwards\_by\_tens = to\_one\_hundred[::-10]  print backwards\_by\_tens |

# Practice Makes Perfect

Great work! See? This list slicing business is pretty straightforward.

Let's do one more, just to prove you really know your stuff.

**1.**

Create a list, to\_21, that's just the numbers from 1 to 21, inclusive.

Create a second list, odds, that contains only the odd numbers in the to\_21 list (1, 3, 5, and so on). Use list slicing for this one instead of a list comprehension.

Finally, create a third list, middle\_third, that's equal to the middle third of to\_21, from 8 to 14, inclusive.

|  |
| --- |
| to\_21 = range(1, 22)  odds = to\_21[::2]  middle\_third = to\_21[7:14] |

**Anonymous Functions**

One of the more powerful aspects of Python is that it allows for a style of programming called **functional programming**, which means that you're allowed to pass functions around just as if they were variables or values. Sometimes we take this for granted, but not all languages allow this!

Check out the code at the right. See the lambda bit? Typing

lambda x: x % 3 == 0

Is the same as

def by\_three(x): return x % 3 == 0

Only we don't need to actually give the function a name; it does its work and returns a value without one. That's why the function the lambda creates is an *anonymous function*.

When we pass the lambda to filter, filter uses the lambda to determine what to filter, and the second argument (my\_list, which is just the numbers 0 – 15) is the list it does the filtering on.

**Instructions**

**1.**

Can you guess what the this code will print to the console? Click Run to see.

|  |
| --- |
| my\_list = range(16)  print filter(lambda x: x % 3 == 0, my\_list) |
| [0, 3, 6, 9, 12, 15] |

**Lambda Syntax**

Lambda functions are defined using the following syntax:

my\_list = range(16) filter(lambda x: x % 3 == 0, my\_list)

Lambdas are useful when you need a quick function to do some work for you.

If you plan on creating a function you'll use over and over, you're better off using defand giving that function a name.

**Instructions**

**1.**

Fill in the first part of the filterfunction with a lambda. The lambdashould ensure that only "Python" is returned by the filter.

Fill in the second part of the filterfunction with languages, the list to filter.

Hint

Remember, filter() takes two arguments: the first is the function that tells it what to filter, and the second is the object to perform the filtering on.

|  |
| --- |
| languages = ["HTML", "JavaScript", "Python", "Ruby"]  # Add arguments to the filter()  print filter(lambda x : x =="Python", languages) |

# Try It!

All right! Time to test out filter() and lambda expressions.

cubes = [x \*\* 3 for x in range(1, 11)] filter(lambda x: x % 3 == 0, cubes)

The example above is just a reminder of the syntax.

**Instructions**

**1.**

Create a list, squares, that consists of the squares of the numbers 1 to 10. A list comprehension could be useful here!

Use filter() and a lambdaexpression to print out only the squares that are between 30 and 70 (inclusive).

Hint

You'll want to filter for x >= 30 and x <= 70.

|  |
| --- |
| squares = [x\*\*2 for x in range(1,11)]  print squares  print filter(lambda x: x>=30 and x <=70, squares ) |

# Iterating Over Dictionaries

First, let's review iterating over a dict.

**Instructions**

**1.**

Call the appropriate method on movies such that it will print out all the items (hint, hint) in the dictionary—that is, each key and each value.

Hint

You'll just want to print the result of calling the .items() method on your movies. No loops necessary!

|  |
| --- |
| movies = {  "Monty Python and the Holy Grail": "Great",  "Monty Python's Life of Brian": "Good",  "Monty Python's Meaning of Life": "Okay"  }  print movies.items() |

**Comprehending Comprehensions**

Good! Now let's take another look at list comprehensions.

squares = [x \*\* 2 for x in range(5)]

**Instructions**

**1.**

Use a list comprehension to create a list, threes\_and\_fives, that consists only of the numbers between 1 and 15 (inclusive) that are evenly divisible by 3 or 5.

Hint

Remember, list comprehension syntax looks like this:

list\_name = [var for var in range]

You can include an optional ifstatement after the range. (You'll need such an if statement to check whether the numbers are evenly divisible by 3 or 5.)

Remember, modulo (%) is a good way to check if a number is evenly divisible by another.

|  |
| --- |
| threes\_and\_fives = [x for x in range(1,16) if x%3==0 or x%5==0] |

**List Slicing**

Great! Next up: list slicing.

str = "ABCDEFGHIJ" start, end, stride = 1, 6, 2 str[start:end:stride]

You can think of a Python string as a list of characters.

**1.**

The string in the editor is garbled in two ways:

1. Our message is backwards.
2. The letter we want is every other letter.

Use list slicing to extract the message and save it to a variable called message.

Hint

It's important to remember that lists are mutable (changeable) in Python, but strings aren't; when you slice a string, you get back a *new string*. The original string is unchanged unless you purposely "save over" it, like this:

my\_string = "Monty Python" # => Monty Python my\_string = my\_string[:-7] # => Monty

|  |
| --- |
| garbled = "!XeXgXaXsXsXeXmX XtXeXrXcXeXsX XeXhXtX XmXaX XI"  message = garbled[::-2] |

# Lambda Expressions

Last but not least, let's look over some lambdas.

my\_list = range(16) filter(lambda x: x % 3 == 0, my\_list)

We've given you another (slightly different) garbled. Sort it out with a filter() and a lambda.

**1.**

Create a new variable called message.

Set it to the result of calling filter()with the appropriate lambda that will filter out the "X"s. The second argument will be garbled.

Finally, print your message to the console.

Hint

Remember, a lambda expression looks like this:

lambda variable: variable expression

For example, you might have

lambda x: x != 10

Remember, filter() takes two arguments: the first is the function that tells it what to filter (in this case, your lambda expression), and the second is the object to perform the filtering on (the garbled string).

|  |
| --- |
| garbled = "IXXX aXXmX aXXXnXoXXXXXtXhXeXXXXrX sXXXXeXcXXXrXeXt mXXeXsXXXsXaXXXXXXgXeX!XX"  message = filter(lambda x : x != "X", garbled)  print message |