LISTening Exercises - Sample Solutions

Christopher Dalziel

November–December 2024

1 Scales, Chords, and List Indexing

A **scale** is a collection of music notes, which are typically selected to sound good together. The most well known scales are the major and minor scales

For this section we will be using the C major scale, which consists of the following notes:

$$C,\ D,\ E,\ F,\ G,\ A,\ B,\ C$$

Note that the second C is played at a higher pitch than the first, so that the scale can be repeated at a higher pitch

1.1 Snippet 1 - Power Chords

A **chord** is a collection of notes played either close together or all at once. The simplest kind of chord is the so called "power" chord, which consists of a **root** (the *first* note), the *fifth*, and an **octave** (the *eighth* note).

When played in sequence, these notes rise in pitch and should sound pleasing to the human ear.

Finish the following snippet using list indexing and concatenation to play the three notes of a C major power chord in order:

```
c_major = ["c", "d", "e", "f", "g", "a", "b", "c"]

# Todo: finish c_major_power
c_major_power = c_major[0] + ???

# Play the notes in c_major_power
for note in c_major_power:
    # Play the given note at the 4th octave for a short period
    play(note, 4)
```

When you've finished this exercise, what do you notice is wrong with the provided code when you run it?

The code should look something like this:

```
# Solution:
c_major_power = c_major[0] + c_major[4] + c_major_[7]
```

The code currently plays the final note at the same pitch as the first note, when the specifications says that the notes should "rise in pitch"

1.2 Snippet 2 - Playing Scales

An **octave** is the gap between a given note (say C_4) and that same note at a higher or lower pitch (C_3 and C_5). In the last exercise the `play()` function defaulted to playing the fourth octave.

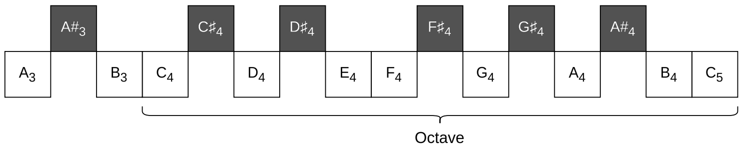

Figure 1: An octave (from C_4 to C_5)

The following code should play through a C major scale, starting from C_4 and ending at C_5. Implement the conditional to increment the octave.

```
c_major = ["c", "d", "e", "f", "g", "a", "b", "c"]
octave = 4

# Play each note in the c_major scale begining at C 4
for index, note in enumerate(c_major):
    # Todo: implement conditional
    if ??? and ???:
        octave += 1
    play(note, octave)
```

If you've never seen the function `enumerate()` before, have a look at the documentation.

Looking at the code you've written and at Figure 1, what could we change about the scale we input that would make our code incorrect?

The expected code is something like this:

```
# Expected:
if note == "c" and index != 0:
    octave += 1
```

Any scale which does contain a C will not increment properly, for example D major:

$$D,\ E,\ F\sharp,\ G,\ A,\ B,\ C\sharp,\ D$$

In general, the note checked for in the condition should be the note in the scale which is nearest above C (or just C), which in D major is $C\sharp$. Writing a solution to do this cleanly is not easy!

```
# Solution:
# Scores for how far a note is from C
score = {"c": 0, "c#": 1, "d": 2, "d#": 3, "e": 4, "f": 5,
↪   "f#": 6, "g": 7, "g#": 8, "a": 9, "a#": 10, "b": 11}

# Make a list of notes paired with their scores
scored = [(note, score[note]) for note in scale]
# Sort the list by the scores and take the lowest score
inc_note = sorted(scored, key=lambda x: x[1])[0][0]

# Play each note in the scale starting at the fourth octave
for index, note in enumerate(scale):
    if note == inc_note and index != 0:
        octave += 1
    play(note, octave)
```

Students should not be expected to understand this code syntactically as it contains many things they are not expected to be familiar with (comprehensions, dictionaries, and lambdas).

Ensure that students understand the fail cases that make the original code incorrect, and the logic underlying the general solution.

2 Constructing Scales with the Circle of Fifths

The **circle of fifths** is an important concept in classical music theory.

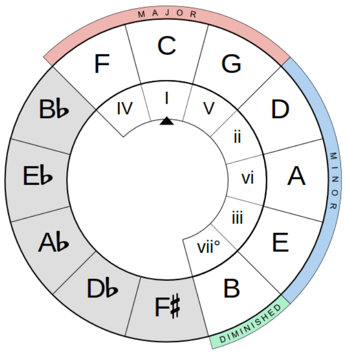

Figure 2: An interactive circle of fifths set to *C* major

Its main use for our purposes is telling us what notes are in a scale. On the example above we can see that the *C* major scale contains the notes *A* through *G* with no sharp or flat notes

Our version of the circle will be implemented as a list. We would represent Figure 4 as follows, with the root note *C* (indicated on the circle with a small triangle) as the first element of the list:

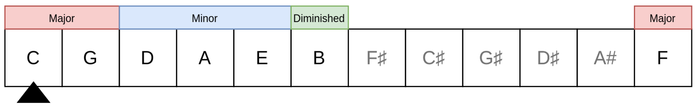

Figure 3: List of fifths set to *C* major

2.1 Snippet 3 - Changing Roots

Thusfar we've been using *C* major as our scale of choice, but suppose we wanted to play a piece in *A* major instead - how would we know what notes were in the scale?

The answer is that we could "rotate" the notes in the circle of fifths so that the little triangle was pointing at *A*, which would be a rotation by 3 notes clockwise ($C \to G \to D \to A$).

You can try this on the interactive circle of fifths by setting the "tonic" column on the left to *A*, it should look like the following (without having been literally rotated - that's editing magic):

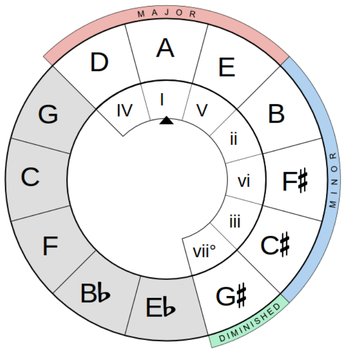

Figure 4: The circle of fifths set to A major

We can implement this idea for our lists using "cycling", an operation which changes the order of our list so that the first n elements (where n is a whole number) get moved to the end of the list.

$$\texttt{cycle}([1,2,3,4,5,6,7,8],3) \rightarrow [4,5,6,7,8,1,2,3]$$

This code snippet should take a list, and return that list with a new first element such that it has been cycled to the left by n steps. Use list slicing to complete the cycle function.

```
list_of_fifths = ["c", "g", "d", "a", "e", "b", "f#", "c#",
↪   "g#", "d#", "a#", "f"]

# Cycles n elements through the provided list
def cycle(list, n):
    # Todo: get the first n elements of list
    start = ???
    # Todo: get the rest of the list
    end = ???

    cycled = end + start
    return cycled
```

How does this cycling function work if we give a negative number for n? Is this what you expected? How can we interpret this in terms of rotation?

The code should look something like this:

```
# Solution:
start = list[:n]
end = list[n:]
```

The cycling function with a positive input is analogous to a clockwise rotation. Giving a negative number cycles the list in a way that is analogous to a counterclockwise rotation.

This is because the negative list indices mean that the slices go from the start to the negative nth position, and from the negative nth element to the end respectively.

2.2 Snippet 4 - Major Construction

When we have the circle of fifths rotated to sit on a root, the notes of the major scale always sit in the same way relative to the root. We can therefore construct a major scale by rotating the circle to the root and taking the notes in those positions.

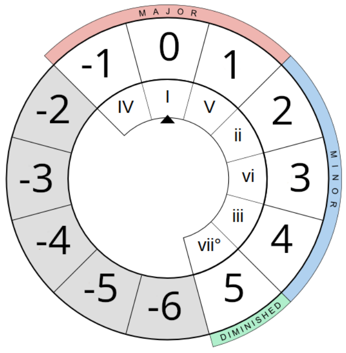

Figure 5: An indexed circle of fifths for major scales

Using this information, complete the following snippet using the `index()` method of lists, list slicing, and negative indexing.

```
list_of_fifths = ["c", "g", "d", "a", "e", "b", "f#", "c#",
↪  "g#", "d#", "a#", "f"]

# Get the major scale for the provided root
def get_major_scale(root):
    # Check that the input is a valid music note
    if not is_music_note(root):
        raise ValueError("Invalid root note")

    # Todo: get the list of fifths set to the root
    cycled = cycle(list_of_fifths, ???)

    # Todo: Get the first 6 elements of the list
    forwards = cycled[:???]
    # Todo: Get the last element of the list
    backwards = cycled[???]

    # Put the notes in alphabetical order
    notes = sorted(forwards + backwards)

    # Return the notes, starting at the root with an extra root
    ↪   on the end
    return cycle(notes, ???) + [root]
```

The code should look something like this:

```
# Get the major scale for the provided root
def get_major_scale(root):
    # Check that the input is a valid music note
    if not is_music_note(root):
        raise ValueError("Invalid root note")

    # Solution:
    cycled = cycle(list_of_fifths,
    ↪   list_of_fifths.index(root))
    forwards = cycled[:6]
    backwards = cycled[-1:]

    # Put the notes in alphabetical order
    notes = sorted(forwards + backwards)

    # Return the notes, starting at the root with an extra
    ↪   root on the end
    return cycle(notes, notes.index(root)) + [root]
```

2.3 Snippet 5 - Relatively Minor

Other scales can be constructed by a similar process (you can see this on the interactive circle by changing the ”mode” column). For example, a minor scale would be from -4 to 2 on Figure 5 instead of -1 to 5.

A major scale’s **relative minor** is a minor scale that contains the same notes as that major scale, but has a different root that makes it “feel” minor.

Converting to minor from major (as shown above) involves subtracting three from the indices on each side, so to get the relative minor we take our root and see what note is three positions to the right.

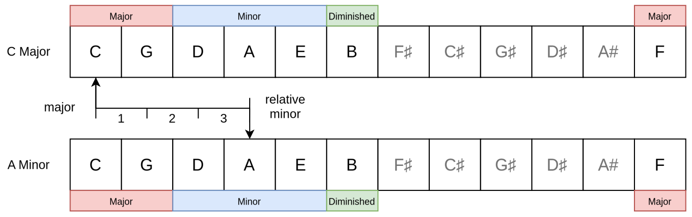

Figure 6: The relative minor of C major is A minor

The following snippet contains a function which should obtain the relative minor, but it throws an error for certain inputs. What inputs does it break for, how might we fix this?

```python
# Get the relative minor for the provided root note
def get_relative_minor(root):
    # Check that the input is a valid music note
    if not is_music_note(root):
        raise ValueError("Invalid root note")

    # The relative minor is three ahead of the given note
    rm_idx = list_of_fifths.index(root) + 3
    relative_minor = list_of_fifths[rm_idx]

    return relative_minor
```

This code fails whenever it tries to look three ahead from $D\sharp$, $A\sharp$, or F. This can be fixed either with modular arithmetic (`rm_idx` should be modulo 11), or via the `cycle()` function:

```
# Get the relative minor for the provided root note
def get_relative_minor(root):
    # Check that the input is a valid music note
    if not is_music_note(root):
        raise ValueError("Invalid root note")

    # The relative minor is three ahead of the given note
    rooted = cycle(list_of_fifths,
    ↪   list_of_fifths.index(root))
    relative_minor = rooted[3]

    return relative_minor
```

3 Putting it all Together

That's the end of the lesson! Congratulations for getting this far. If you've still got some time, there's one final activity that you can do - if you've completed the exercises thus-far this one shouldn't be too difficult :)

Using what you've done so far, write a program that plays any requested major scale followed by its relative minor scale. Feel free to copy and edit code from the previous exercises to complete this task.

A sample solution for this final excercise:

```python
list_of_fifths = ["c", "g", "d", "a", "e", "b", "f#", "c#",
↪   "g#", "d#", "a#", "f"]
# Adapted from snippet 2
def play_scale(scale, octave):
    # Scores for how far a note is from C
    score = {"c": 0, "c#": 1, "d": 2, "d#": 3, "e": 4, "f":
    ↪   5, "f#": 6, "g": 7, "g#": 8, "a": 9, "a#": 10, "b":
    ↪   11}

    # Make a list of notes paired with their scores
    scored = [(note, score[note]) for note in scale]
    # Sort the list by the scores and take the lowest score
    inc_note = sorted(scored, key=lambda x: x[1])[0][0]

    for index, note in enumerate(scale):
        if note == inc_note and index != 0:
            octave += 1
        play(note, octave)

def play_major_and_relative_minor(root):
    # Get the major scale (snippet 4)
    major = get_major_scale(root)
    # Get the relative minor and construct the scale
    ↪   (snippet 5)
    relative_minor = get_relative_minor(root)
    minor = cycle(major[0:-1], major.index(relative_minor))
    ↪   + [relative_minor]

    # Play the scales
    play_scale(major, 4)
    sleep(1)
    play_scale(minor, 3)

play_major_and_relative_minor("c")
```