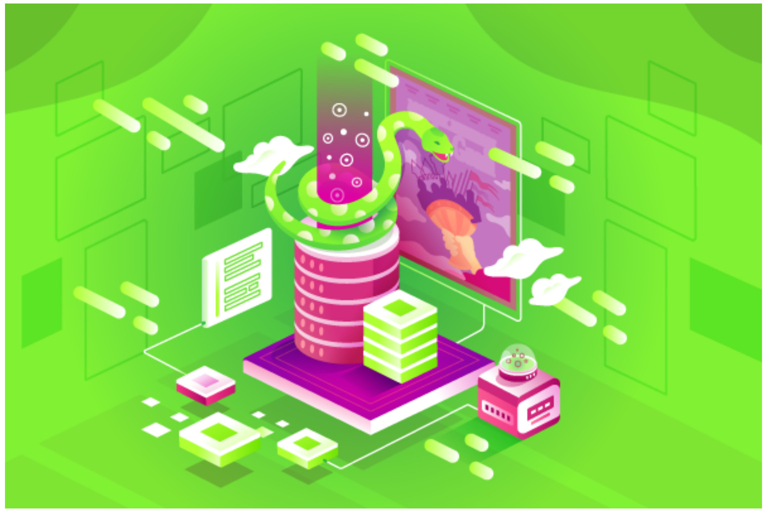
Study Guide



Certified Entry-Level Python Programmer Certification Study Guide

Keith Thompson
keith@linuxacademy.com
Nov 12, 2019

Linux Academy

Contents

Basic Concepts (17% - 5 Exam Items)

Fundamental Concepts

Interpreting and the Interpreter

Python is an interpreted programming language, meaning that segments of code can be read and immediately executed by the `python` executable.

Compilation and the Compiler

The compilation is the process of converting from one language to another. This often means from a higher-level programming language into a more machine-readable language. In the case of Python, code is compiled from the Python source code into Python byte code that can be run by the Python virtual machine.

Syntax and Semantics are the rules of a programming language.

Python Keywords

```
|       |      |      |     |    |        |       |
|-------|------|------|-----|----|--------|-------|
| False | None | True | and | as | assert | async |
| await | break | class | continue | def | del | elif |
| else | except | finally | for | from | global | if |
| import | in | is   | lambda | nonlocal | not | or |
| pass | raise | return   | try   | while   | with   | yield |
```

Indenting

Python relies on indentation to separate blocks of code. The community accepted indentation is 4 spaces per nested context.

Literals

Booleans

There are only 2 different possible Boolean values representing true and false. In Python they are:

- `True`
- `False`

Integers

Integers are positive and negative, non-decimal numbers. Examples:

- `-1`
- `10`
- `1e5` - Scientific Notation
- `1E-1` - Scientific Notation

Floating-Point Numbers

Floating-point numbers are positive and negative numbers with a fractional component. Examples:

- `-1.0`
- `10.5`
- `10.5e2` - Scientific Notation
- `11.5E-1` - Scientific Notation

Strings

Strings represent textual data and are a collection of characters surrounded by quotes (either single or double). Examples:

- `'Testing'`
- `"Test"`
- `'Password123!@#$'`
- `"New\nLine"` - Including escape character `'\n'`

Multiline strings are created using triple quotes at the beginning and end:

```
"""This is
a Multi-line
string"""

'''
Also a multi-line
string
'''
```

We can place quotes within a string by either using the oppose quotes on the boundaries or by escaping the quote using a backslash:

```
"Using 'single' in doubles"
'Using "doubles" in singles'
"Using \"doubles\" in doubles"
'Using \'singles\' in singles'
```

Useful escape characters:

- `\n` - Newline character
- `\t` - Tab character
- `\\` - Inserts a backslash into the string instead of escaping the next character.

Comments

Comments are lines, or the trailing end of a line, that are not executed and are ignored by the interpreter. All comments in Python start with a `#` character. There are no block comments in Python. Example comments:

```
# This is a whole line comment
x = 1 # This is a trailing comment
```

The `print()` Function

The `print` function is a built-in function that is always available in Python and is used to write messages to the screen.

The `input()` Function

The `input` function is a built-in function that can prompt the user of the code to write text and stores that text after the Enter/Return key is hit.

Numeral Systems (binary, octal, decimal, hexadecimal)

Every integer number can be represented in multiple ways depending on the numeral system. The standard numeral system use in everyday life is *decimal*, meaning that it is a base 10 system. It gives us 10 unique digits per position in a number (0-9). Other number systems exist, and have different numbers of possible digits. Here are other common numeral systems and the available digits:

- binary - 2 digits (0-1)
- octal - 8 digits (0-7)
- decimal - 10 digits (0-9)
- hexadecimal - 16 digits (0-9 and A-F)

Numeric Operators

Operators are special symbols (or multiple symbols) that take one or two operands and perform an operation. Here are the standard numeric operators:

- `+` - Addition
- `-` - Subtraction
- `**` - Exponentiation
- `*` - Multiplication
- `/` - Floating-point division (standard division)
- `//` - Floor division (returns integer result without a remainder)
- `%` - Modulus (returns the integer remainder from division)

String Operators

Types besides numeric types also have operators available to them. For string, there are two available operators:

- `+` - Concatenation: Combines two strings and returns a new string
- `*` - Multiplication: Repeats and concatenates the string multiple times

Data Types, Evaluations, and Basic I/O Operations (20% - 6 Exam Items)

Operators

Unary and Binary Operators

Operators require operands (the arguments on the right and left of the operator). Unary operators only take a single operand to the right of the operator. Operators that take two operands are "binary" operators.

Operator Priority and Binding

To know how to execute a line of code with multiple operators there needs to be an order of operations that defines which operators should be executed first. This is called operator priority, or operator binding. Here's the priority order for operators, literals, and functions from highest priority (occurs first) to lowest priority (occurs last):

- Parenthesis and List/Dictionary/Set literals
- Accessing attributes (subscription, slicing, function/method call, attribute reference)
- Exponentiation (`**`)
- Positive, Negative, and bitwise complement
- Multiplication `*`, Division `/`, Floor Division `//`, Modulo `%`
- Addition `+`, Subtraction `-`
- Bitwise Shifts `<<` & `>>`

- Bitwise AND `&`
- Bitwise XOR `^`
- Bitwise OR `|`
- Comparison operators (`in`, `not in`, `is`, `is not`, `<`, `>`, `<=`, `>=`, `==`, `!=`)
- Boolean NOT `not`
- Boolean AND `and`
- Boolean OR `or`
- Conditions `if`

Bitwise Operators

Bitwise operators are operators that work off of the bit information (binary notation) for numbers. Positions in binary numbers are known as *bits* and they can either be `0` or `1`. Bitwise operations do various things based on the values of these bits. Here are the bitwise operators with examples of how they work:

Bitwise Complement (`~`)

The bitwise complement is a unary operator that essentially processes the number (`x`) through the equation `-x - 1`:

```
>>> x = 0b010
2
>>> ~x
-3
>>> bin(~x)
'-0b11`
```

Bitwise OR (|)

The bitwise *OR* operator takes two binary numbers and returns a binary number where each position in the operands is compared to the same position in the other operand. If *either* are `1`, then the result will have a `1` in that position; otherwise the result will have a `0` in that position. Here's an example:

```
>>> a = 0b1001
>>> b = 0b1100
>>> bin(a | b)
'0b1101'
```

Bitwise AND (`&`)

The bitwise *AND* operator takes two binary numbers and returns a binary number where each position in the operand is compared to the same position in the other operand. If *both* are `1`, then the result will have a `1` in that position; otherwise the result will have a `0` in that position. Here's an example:

```
>>> a = 0b1001
>>> b = 0b1100
>>> bin(a & b)
'0b1000'
```

Bitwise XOR (`&`)

The bitwise *XOR* operator takes two binary numbers and returns a binary number where each position in the operand is compared to the same position in the other operand. If *exactly one* is `1` then the result will have a `1` in that position; otherwise the result will have a `0` in that position. Here's an example:

```
>>> a = 0b1001
>>> b = 0b1100
>>> bin(a ^ b)
'0b101'
```

Bitwise Right Shift (>>)

The bitwise shift operators allow us to shift our bit values directly sideways by a certain number of positions. To shift our bits to the right we'll use the bitwise right shift operator which is `>>`. Our initial values are on the left-hand side and the number of positions to shift is on the right:

```
>>> a = 0b110
>>> bin(a >> 2)
'0b1'
>>> bin(a >> 4)
'0b0'
```

Notice that if we shift beyond the number of bits in our number then we simply get `0` as the result.

Bitwise Left Shift (<<)

Bitwise Left Shift uses the `<<` operator with the same rules as the right shift operator. For each position that we shift, we'll add a new `0` bit to the right.

```
>>> a = 0b110
>>> bin(a << 2)
'0b11000'
>>> bin(a << 4)
'0b1100000''
```

Boolean Operators

The `not` Operator

The `not` operator is a unary operator that takes any object as an operand and returns the opposite Boolean value:

```
>>> not True
False
>>> not False
True
>>> not 1
False
>>> not 0
True
```

Remember that every object has a Boolean representation. Zero values, `None`, and empty sequences are all considered `False` and all other values are `True`.

The `or` Operation

The Boolean *or* operator works the same way that the bitwise *OR* operator did if we are only considering one bit. The bit of `1` is equivalent to `True` and `0` is equivalent to `False`:

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
```

```
>>> False or True
True
```

The `and` Operation

The `and` operator is the opposite of `or` and both of the operands need to be true:

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
>>> False and True
False
```

Relational (Comparison) Operators

Relational operators allow us to make comparisons between two different objects. Relational operators always return a Boolean result (unless there is an error from comparing incomparable types). Here are the relational operators:

- `==` - Equal operator: Returns `True` if the operands are equivalent. The equivalence allows floats and integers to be compared and return `True` for `1 == 1.0`.
- `!=` - Not equal operator: Returns `True` if the operands are not equivalent.
- `>` - Greater than operator: Returns `True` if the left-hand operand is greater than the right-hand operand.
- `>=` - Greater than or equal to operator: Returns `True` if the left-hand operand is greater than or equal to the right-hand operand.
- `<` - Less than operator: Returns `True` if the left-hand operand is less than the right-hand operand.

- `<=` - Less than or equal to operator: Returns `True` if the left-hand operand is less than or equal to the right-hand operand.

Floating-Point Accuracy

Not all decimal numbers that we can write out can be represented by a computer as a float. The reason for this is that floats are stored on computer hardware as binary fractions, and not all decimal (base 10 numbers) can be represented as binary fractions. This leads to the floating-point numbers that we work with actually being approximations of the binary fraction representation.

An example of this is the decimal number `0.1` which has no binary fraction equivalent. Depending on the machine, the exact approximation used behind the scenes may be different. But since Python 3.1, the representation that is returned to the user is the clean number even though the number used behind the scenes is something like `0.1000000000000000055511151231257827021181583404541015625`.

Type Casting

To be able to convert from one type to another we will use type casting. To cast from one type to another we will pass a value of one type to the initializer function of the destination type. Here are examples:

```
>>> int("1")
1
>>> str(1.0)
'1.0'
>>> float('1.2')
1.2
>>> bool('Hello')
True
```

More About the `print` Function

Taking multiple arguments

The `print` function is a variadic function that can take a variable number of arguments to print out along with some keyword arguments. When called with multiple value arguments, a space will be added between the values when printed out.

```
>>> print('Hello', 'there', 'Billy')
Hello there Billy
```

The `sep` Parameter

The string inserted between each value can be customized by providing the keyword argument `sep` with the string to use instead of the default space.

```
>>> print('Hello', 'there', 'Billy', sep=',')
Hello,there,Billy
```

The `end` Parameter

By default, there is a newline character added to the end of the message printed by the `print` function. This can be demonstrated by calling `print` multiple times, and seeing that each call prints on a separate line. Consider this code:

```
print('Hello')
print('There')
```

It would output the following when run:

```
Hello
There
```

The keyword argument `end` can be provided with a value to use at the end, instead of the default newline character. Look at this code:

```
print('Hello', end=' ')
print('There')
```

It will output the following when run:

```
Hello There
```

String Indexing, Slicing, and Immutability

String Immutability

Strings are an immutable sequence type. This means that after a string has been created, Python will always reference the same location in memory when it needs that specific string value. It also means that we cannot change a string, only make new strings.

Indexing Strings

To read the character from a specific position in a string, we can use indexing. The index must exist in the string or an `IndexError` will be raised. Indexing will return a different string that is only 1 character:

```
>>> 'test string'[2]
's'
```

String Slicing

To get a substring from a string, we can use slicing. A slice is comprised of a starting index, an ending index, and a step value all separated by colon characters. The result of slicing will return a new string of varying length.

```
>>> 'test string'[1:3]
'es'
>>> 'test string'[:3]
'tes'
>>> 'test string'[:]
'test string'
>>> 'test string'[4:8:2]
' t'
>>> 'test string'[::2]
'ts tig'
>>> 'test string'[::-1]
'gnirts tset'
```

Flow Control - Loops and Conditional Blocks (20% - 6 Exam Items)

Conditional Statements

The `if` Statement

When we would only like to run a specific block of code if a condition is met, then we need to use an `if` statement.

An `if` statement is constructed like this: The conditional body will only be run if the result of the `CONDITION` is `True` when implicitly converted to a bool:

```
if CONDITION:
    print('Conditional body')
```

The `else` Clause

If we would like to run a block of code when we do not execute the body of an `if` statement, then we can add an `else` clause. The `else` needs to be at the same level of indentation as the `if`, should be the final clause in the conditional, and only gets run if no other branches of the conditional were.

```
if CONDITION:
    print("if body")
else:
    print("else body")
```

The `elif` Clause

The `elif` clause allows us to evaluate additional conditions in our `if` statements. We can add any number of `elif` clauses, each with a condition. Just like with the `if` statement's condition, if the condition of an `elif` clause evaluates to `True`, then the body of that clause will be run. When the body of any branch in the conditional is run, then the evaluation of the remaining `elif` clauses is skipped.

```
if CONDITION1:
    print('if body')
elif CONDITION2:
    print('elif 1 body')
elif CONDITION3:
    print('elif 2 body')
else:
    print('else body')
```

The `pass` Statement

Occasionally we want to create a new code context like an `if` statement, but don't want to write the code in the body just yet. Since `if` statements and loops can't be created without a body, we need to have at least one executable line within them. The `pass` statement allows us to write a line of code that does absolutely nothing. This is called a "noop" (no operation).

```
if a < b:
    pass
```

Building Loops

The `while` Loop

The `while` loop is the most basic type of loop that evaluates a condition, and if it is *true* then the body of the loop will execute and the condition will be re-evaluated:

```
while CONDITION:
    print("print something")
```

If no aspect of the condition is changed from within the loop body, then the loop will continue forever. For this reason, we'll usually want to use a variable in our condition, and then make a change to the variable:

```
count = 1
while count < 10:
    print(count)
    count += 1
```

This will print `1` through `9` before the condition evaluates to false and the loop stops.

The `for` Loop

The most common use we have for looping is when we want to execute some code for each item in a sequence. For this type of looping or iteration, we'll use the `for` loop. The general structure for a `for` loop is:

```
for TEMP_VAR in SEQUENCE:
    pass
```

The `TEMP_VAR` will be populated with each item as we iterate through the `SEQUENCE` and it will be available to us in the context of the loop. After the loop finishes one iteration, then the `TEMP_VAR` will be populated with the next item in the

`SEQUENCE`, and the loop's body will execute again. This process continues until we either hit a `break` statement, or we've iterated over every item in the `SEQUENCE`. Here's an example of looping over a list of colors:

```
colors = ['blue', 'green', 'red', 'purple']
for color in colors:
    print(color)
```

The `range` Type

Ranges are an immutable sequence type that define a start, a stop, and a step value, and then the values within are calculated as it is interacted with. This allows for ranges to be used in place of sequential lists, and takes less memory and includes more items:

```
>>> my_range = range(10)
>>> my_range
range(0, 10)
>>> list(my_range)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 14, 2))
[1, 3, 5, 7, 9, 11, 13]
```

Ranges can also be used in a `for` loop without converting to a list if we want to repeat something a set number of times:

```
>>> for _ in range(1, 5):
...     print("looping")
...
looping
looping
looping
```

```
looping
>>>
```

Nesting Loops and Conditionals

We're able to nest conditionals and loops in Python. To do this, we need to indent each nested context four more spaces than the parent context. Here's an example of a conditional nested within a loop:

```
while counter <= 25:
    if counter % 4 == 0:
        print(counter)
    counter += 1
```

Controlling Loop Execution

It is possible to manipulate the execution of a loop so that the current execution is skipped and the next iteration is started, or so that the entire loop exits prematurely. To do this, we use the `continue` and `break` statements. These statements can be used with both `while` and `for` loops.

Here's an example of the `continue` statement:

```
>>> count = 0
>>> while count < 10:
...     if count % 2 == 0:
...         count += 1
...         continue
...     print(f"We're counting odd numbers: {count}")
...     count += 1
...
We're counting odd numbers: 1
We're counting odd numbers: 3
```

```
We're counting odd numbers: 5
We're counting odd numbers: 7
We're counting odd numbers: 9
>>>
```

Here's an example of the `break` statement:

```
>>> count = 1
>>> while count < 10:
...     if count % 2 == 0:
...         break
...     print(f"We're counting odd numbers: {count}")
...     count += 1
...
We're counting odd numbers: 1
```

The `else` Clause on Loops

The `else` clause for loops in Python allows us to define an additional code context that will execute when the loop has naturally finished its iteration. This means for a `for` loop that we've reached the end of whatever we were iterating through, and for the `while` loop the conditional has evaluated to `False`. The `else` clause's body *will not* run if the loop stops because of a `break` clause. This `else` clause will *always* run:

```
a = 1
while a < 10:
    print(a)
    a += 1
else:
    print("else body")
```

This `else` clause would not run because of the `break` statement:

```
a = 1
while a < 10:
    print(a)
    if a % 5 == 0:
        break
    a += 1
else:
    print("else body")
```

Data Collections - Lists, Tuples, and Dictionaries (23% - 7 Exam Items)

Lists

Constructing a list

List literals are constructed using square brackets (`[]`) and each item in the list is separated by a comma (`,`). Python lists can contain multiple different types of objects, so they don't all need to be strings, or integers.

```
my_list = ['a', True, 1, 1.0]
```

Indexing and Slicing

Just like strings, items are accessed in lists using indexing and slicing. Indexing will return the item held at the specified index. Slicing will return a new list containing 0 or more items from the original list:

```
>>> my_list = ['a', True, 1, 1.0]
>>> my_list[1]
True
>>> my_list[-1]
1.0
>>> my_list[2:]
[1, 1.0]
```

Because lists are mutable we can also use indexing and assignment together to change the items in the list:

```
>>> my_list[0] = 1
>>> my_list
[1, True, 1, 1.0]
```

The `in` Operator

To check if an item is in a collection, we can use the `in` operator:

```
>>> 2 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
```

We can also determine if something is not in a collection by using the `not in` operator:

```
>>> 2 not in [1, 2, 3]
False
>>> 4 not in [1, 2, 3]
True
```

The `in` and `not in` operators work for strings, tuples, lists, and dictionaries.

The `len` Function

To determine the length of a collection, like a list or a string, we can use the `len` function. The `len` function will return the number of items in the collection. In the case of indexed data types, like lists or strings, then the value returned by the `len` function will always be 1 higher than the index of the last item or character:

```
>>> len(['a', 'b', True, False])
4
>>> len('Testing')
7
```

The `append` Method

The list's `append` method allows us to add an item to the end of an existing list. Since lists are mutable, this will change the list itself and the variables that point to the list:

```
>>> my_list = [1, 2, 3]
>>> my_list.append(4)
>>> my_list
[1, 2, 3, 4]
```

The `insert` Method

The `insert` method allows us to place an item into an existing list at a specified index. All of the items that originally had an index greater than or equal to the index we're inserting the new item at will have their indexes increased:

```
>>> my_list = [1, 2, 3]
>>> my_list.insert(0, 'a')
>>> my_list
['a', 1, 2, 3]
>>> my_list.insert(2, 'b')
>>> my_list
['a', 1, 'b', 2, 3]
```

The `index` Method

The `index` method will return the index of an item if it exists within the list. If the item is not in the list, then an error will be raised:

```
>>> my_list = [1, 2, 3]
>>> my_list.index(2)
1
>>> my_list.index('a')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 'a' is not in list
```

The `sorted` Function

The `sorted` function takes a collection, converts it to a list, and then returns a new list with the values sorted:

```
>>> sorted([1, 4, 3, 2])
[1, 2, 3, 4]
>>> sorted('Testing')
['T', 'e', 'g', 'i', 'n', 's', 't']
```

The `reversed` Function

The `reversed` function takes a list and reverses its order. It does not return a list. It returns an object that can be converted back into a list. Note that it does not change the original list:

```
>>> my_list = [1, 3, 4]
>>> reversed(my_list)
<list_reverseiterator object at 0x102af3b50>
>>> list(reversed(my_list))
```

```
[4, 3, 1]
>>> my_list
[1, 3, 4]
```

The `del` Statement

The `del` statement allows us to delete items from collections, and also to completely delete variables. To delete an item from a list or dictionary, use the `del` statement combined with the indexing or keying operation that you would use to access the item that you'd like to delete from the collection:

```
>>> my_list = [1, 2, 3]
>>> del my_list[0]
>>> my_list
[2, 3]
>>> ages = {'Kevin': 60, 'Bob': 40}
>>> del ages['Kevin']
>>> ages
{'Bob': 40}
```

List Comprehensions

List comprehensions allow us to iterate through a list, performing an action on items from the list and returning a new list with the results of those actions. And it's all done in a single statement. A list comprehension can be used to replace a `for` loop like this:

```
my_list = ['a', 'b', 'c']
upper = []
for l in my_list:
    upper.append(l.upper())
```

Here's this same thing as a list comprehension:

```
my_list = ['a', 'b', 'c']
upper = [l.upper() for l in my_list]
```

The result of the code to the left of the `for` will be placed into the new list.
We can also provide a condition so that we can skip certain items in the original list:

```
odds = [num for num in range(1, 11) if num % 2 == 1]
# odds is [1, 3, 5, 7, 9]
```

Nested Lists (Matrices)

Matrices are a structure that has rows and columns. To model a matrix in Python, we need a new list for each row, and we'll need to make sure that each list is the same length so that the columns work properly.

Here's an example matrix not in code:

```
1 2 3
4 5 6
```

To model this matrix in Python, we'll do this:

```
>>> my_matrix = [[1, 2, 3],
...              [4, 5, 6]]
>>> my_matrix
[[1, 2, 3], [4, 5, 6]]
```

To determine how many rows are in a multi-dimensional list, we need to use the `len` function on the matrix itself. And to get the number of columns, we would use `len` on any row in the matrix (assuming that it's a proper matrix with each row having the same number of columns):

```
>>> row_count = len(my_matrix)
>>> column_count = len(my_matrix[0])
>>> row_count
2
>>> column_count
3
```

Cubes and Squares

Matrices that have the same number of columns as rows are known as "cubes". If a matrix is 2x2 then it is also known as a "square". A "4-cube" is a 4x4 matrix, a "5-cube" is 5x5, etc.

Tuples

Working with Tuples

A tuple is a fixed width, immutable sequence type. We create tuples using parenthesis (`()`) and at least one comma (`,`):

```
>>> point = (2.0, 3.0)
```

Since tuples are immutable, we don't have access to the same methods that we do on a list. We can use tuples in some operations like concatenation, but we can't change the original tuple that we created:

```
>>> point_3d = point + (4.0,)
>>> point_3d
(2.0, 3.0, 4.0)
```

Tuples can be indexed and sliced just like lists, except that the result of the slice will also be a tuple:

```
>>> large_tuple = (1, 2, 3, 4, 5, 6)
>>> large_tuple[2]
```

```
3
>>> large_tuple[3:5]
(4, 5)
```

Tuples vs. Lists

When determining whether we should use a list or a tuple, we need to ask ourselves one important question:

> Will we ever not know the exact number of items that we're storing?

If we answer *yes* to this question, then we should use a list. Lists are great for holding onto real collections: users, phone numbers, etc.

Tuples make more sense in two general situations: 1. We're trying to return more than one piece of information from a function. 2. We want to model something with a specific number of fields that we can positionally hold in a tuple.
- A point in 2d or 3d space has `x` and `y` coordinates (and potentially `z`), and those values should always be in a specific spot.

Dictionaries

Creating Dictionaries

Dictionaries are the main mapping type that we'll use in Python. This object is comparable to a Hash or *associative array* in other languages.

We create dictionary literals by using curly braces (`{` and `}`), separating keys from values using colons (`:`), and separating key/value pairs using commas (`,`). Here's an example dictionary:

```
>>> ages = { 'kevin': 59, 'alex': 29, 'bob': 40 }
>>> ages
{'kevin': 59, 'alex': 29, 'bob': 40}
```

Accessing Dictionary Items

Accessing items in a dictionary looks like indexing a list, except instead of an index we use the key:

```
>>> ages['kevin']
59
```

Attempting to access a key that is not in the dictionary will result in an error:

```
>>> ages['David']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'David'
```

Because dictionaries are mutable, we can also pair assignment with keying into the dictionary to change the value for the specified key. You can also do this for a key that doesn't exist to add the key/value pair:

```
>>> ages['kevin'] = 60
>>> ages['David'] = 30
>>> ages
{'kevin': 59, 'alex': 29, 'bob': 40, 'David': 30}
```

The `get` Method

To avoid an error when attempting to access a key from a dictionary, we can check if the key exists by using the `in` operator:

```
>>> 'Alice' in ages
False
```

An alternative is to use the `get` method, which will return `None` if the key does not exist in the dictionary:

```
>>> ages.get('Alice')
>>> ages.get('kevin')
59
```

The `keys` Method

The dictionary type's `key` method will return an iterable type with all of the keys from the dictionary. The `dict_keys` type can be converted to a list:

```
>>> ages.keys()
dict_keys(['kevin', 'alex', 'bob', 'David'])
>>> list(ages.keys())
['kevin', 'alex', 'bob', 'David']
```

The `values` Method

The dictionary type's `key` method will return an iterable type with all of the values from the dictionary. The `dict_values` type can be converted to a list:

```
>>> ages.values()
dict_values([59, 29, 40, 30])
>>> list(ages.values())
[59, 29, 40, 30]
```

The `items` Method

The dictionary type's `key` method will return an iterable type with all of the key/value pairs from the dictionary as tuples. The `dict_items` type can be converted to a list:

```
>>> ages.items()
dict_items([('kevin', 59), ('alex', 29), ('bob', 40), ('David', 30)])
>>> list(ages.items())
[('kevin', 59), ('alex', 29), ('bob', 40), ('David', 30)]
```

The `clear` Method

The dictionary type has a `clear` method that will remove all of the key/value pairs from the dictionary.

```
>>> ages.clear()
>>> ages
{}
```

String Encodings, Methods, and Functions

String Encodings

In Python 3, strings are Unicode by default (specifically UTF-8 encoded). This means that behind the scenes, each character is stored as its Unicode code point, but what does that mean? Unicode is an encoding standard that allows all sorts of unique characters across different languages to have consistent unique numeric values. For instance, the code point for the letter `a` is `97`, but the character that isn't commonly typed out like the trademark symbol `™` has a code point of `8482`. Unicode is a standard, but there are different encodings with one of the most commonly used being UTF-8. UTF-8 stands for "Unicode Transformation Format" with the "8" meaning that values are 8-bits in length. 1,112,064 valid Unicode code points can be encoded with UTF-8.

Up to this point, we've been working with strings that use standard ASCII characters (letters, numbers, and common punctuation). ASCII is an older specification that named 256 characters and their corresponding code points and the first 128 are valid UTF-8 values also. The trademark symbol is in the ASCII specification under the extended characters table, so it has an ASCII numeric value between 128 - 255, but a completely different Unicode code point.

To make things even more confusing, Unicode code points are sometimes represented in decimal form, and other times by using a 'U+' followed by the hexadecimal form of the same number. So '™' is `8482`, but if you search the internet you'll usually see it written as `U+2122`.

The `ord` and `chr` Functions

To convert from character to the Unicode code point we can use the `ord` function:

```
>>> ord('l')
108
```

If we already know the Unicode code point for a character that is more difficult to type, like the trademark symbol, then we can type it out using the hexadecimal notation if we prefix the number with `\u`:

```
>>> ord('\u2122')
8482
```

To convert from a Unicode code point integer back to a string we can use the `chr` function:

```
>>> '\u2122'
'™'
>>> '\u2124'
'ℤ'
```

Useful String Functions

- `upper` - Converts the string to all uppercase:

```
'TeSt'.upper() # => 'TEST'
```

- `lower` - Converts the string to all lowercase:

```
'TeSt'.lower() # => 'test'
```

- `capitalize` - Capitalized the first character and lowercases the rest:

```
'teSt'.capitalize() # => 'Test'
```

- `title` - Capitalizes the first character of each word in the string:

```
'this is a test'.title() # => 'This Is A Test'
```

- `split` - Creates a list of strings by separating a string on a specific string:

```
'this is a test'.split() # => ['this', 'is', 'a', 'test']
'Hello,There,Kevin'.split(',') # => ['Hello', 'There', 'Kevin']
```

- `join` - Inserts the object that we're calling the method on between each string in the list of strings passed to the method:

```
', '.join(['Hello', 'There', 'Kevin']) # => 'Hello, There, Kevin'
```

The `isxxx` Methods

Since strings are collections of characters they can hold onto an incredible variety of information, but that doesn't mean that there aren't different patterns that the information could fall into. For instance, the string `'12'` is completely numeric. For these types of patterns the `str` class provides a whole family of methods that start with `is`, such as `isnumeric`:

```
>>> "12".isnumeric()
True
```

There are quite a few of these methods, here's a list: * `isascii` - Return True if all characters in the string are ASCII, False otherwise. * `islower` - Return True if the string is lowercase, False otherwise. * `isupper` - Return True if the string is uppercase, False otherwise. * `istitle` - Return True if the string is title-cased (all words capitalized), False otherwise. * `isspace` - Return True if the string is a whitespace string, False otherwise. * `isdecimal` - Return True if the string is a decimal string (whole number), False otherwise. * `isdigit` - Return True if the string is a digit string (whole number), False otherwise. * `isnumeric` - Return True if the string is numeric (whole number), False otherwise. * `isalpha` - Return True if the string is alphabetic, False otherwise. * `isalnum` - Return True if the string is an alphanumeric string, False otherwise. * `isidentifier` - Return True if the string is a valid Python identifier, False otherwise. The string could be used as a variable, function, or class name. * `isprintable` - Return True if the string is printable, False otherwise. If the character can't be printed as is, then it's not printable. So escape characters like `\n` are considered not printable, even though they change how the string is printed.

Functions (20% - 6 Exam Items)

Defining and Invoking Functions

We can create functions in Python using the following:

- The `def` keyword
- The function name - lowercase starting with a letter or underscore (`_`)
- Left parenthesis (`(`)
- 0 or more parameter names
- Right parenthesis (`)`)
- A colon `:`
- An indented function body

Here's an example without any parameters:

```
>>> def hello_world():
...     print("Hello, World!")
...
>>> hello_world()
Hello, World!
>>>
```

If we want to define a parameter we will put the variable name we want it to have within the parentheses:

```
>>> def print_name(name):
...     print(f"Name is {name}")
...
>>> print_name("Keith")
Name is Keith
```

The `return` Keyword

When we want a function to return a value, instead of performing a side-effect like writing to the screen, then we need to use the `return` statement. When a `return` statement is hit during a function's execution, the value passed to the statement is returned to the caller and the remainder of the function is not executed.

```
>>> def add(num1, num2):
...     return num1 + num2
...
>>> my_val = add(1, 3)
>>> my_val
4
```

If no `return` statement is hit by the time the function has fully executed, then the function effectively returns `None`. And if there is a `return` statement without any values passed to it, then it is essentially `return None`.

This function:

```
def get_item(my_list, item):
    if item in my_list:
        index = my_list.index(item)
        return my_list[index]
    return

my_list = [1, 2, 3, 4]
```

```
get_item(my_list, 2) # => 2
get_item(my_list, 15) # => None
```

Is essentially:

```
def get_item(my_list, item):
    if item in my_list:
        index = my_list.index(item)
        return my_list[index]

my_list = [1, 2, 3, 4]
get_item(my_list, 2) # => 2
get_item(my_list, 15) # => None
```

The `None` Keyword

The `None` keyword represents nothingness in Python. A variable can't exist without some sort of value. But when there is no useful value for a variable to have we can use the `None` keyword, which is the only instance of the `NoneType` class.

Recursion

Recursion is the practice of calling a function from within itself. This might not seem like something that you'd ever do at first, but occasionally problems can be solved by breaking up the problem into smaller versions of the same problem. For recursion to work, we need one or more *base cases*. In a base case, the function needs to return something and not call itself. Without this, the function will never return and continue to call itself forever.

Here's an example of a recursive function that calculates the nth number in the Fibonacci Sequence:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1

    return fib(n - 2) + fib(n - 1)
```

Recursion has its limits. The main issue with recursion is that every time we recurse, we're adding more and more function calls to the stack of calls that need to be completed. Some languages are optimized to handle this by implementing something called "tail-call optimization", but Python is not one of those languages. Recursion is a useful tool at times, but it does require being delicate and layering in some manual optimization.

Parameters vs. Arguments

While often used interchangeably, there is a difference between parameters and arguments when we're talking about functions. Parameters are the variables within the definition of the function and the function body. Arguments are the *values* passed into a function when the function is called.

Keyword and Position Arguments

Every function call we've made up to this point has used what are known as positional arguments. But if we know the name of the parameters, and not necessarily the positions, we can all them *all* using keyword arguments like so:

```
>>> def contact_card(name, age, car_model):
...     return f"{name} is {age} and drives a {car_model}"
...
>>> contact_card("Keith", 29, "Honda Civic")
'Keith is 29 and drives a Honda Civic'
>>> contact_card(age=29, car_model="Civic", name="Keith")
'Keith is 29 and drives a Civic'
>>> contact_card("Keith", car_model="Civic", age="29")
'Keith is 29 and drives a Civic'
>>> contact_card(age="29", "Keith", car_model="Civic")
```

```
File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

When we're using position and keyword arguments, every argument after the first keyword argument *must* also be a keyword argument. It's sometimes useful to mix them, but oftentimes we'll use either all positional or all keyword.

Parameters with Default Arguments

Along with being able to use keyword arguments when we're calling a function, we're able to define default values for parameters to make them optional when the information is commonly known and the same. To do this, we use the assignment operator (`=`) when we're defining the parameter:

```
>>> def can_drive(age, driving_age=16):
...     return age >= driving_age
...
>>> can_drive(16)
True
>>> can_drive(16, driving_age=18)
False
```

Parameters with default arguments need to go at the end of the parameters list when defining the function, so that positional arguments can still be used to call the function.

Once we've set a default argument for a parameter, then every parameter to the right of the one with a default must also have a default. Otherwise, we'll get a `SyntaxError`:

```
>>> def can_drive(age, driving_age=16, vehicle_type):
...     pass
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Defining and Using Generators

A generator is a function that behaves like an iterator. This means that we can ask a generator function for its "next" value, and it will calculate it and return it to us. Similar to how a `range` doesn't calculate all of the values at once, a generator function essentially *pauses* its execution after returning a single result, until the next result is requested.

Generator functions are defined the same way that traditional functions are, except that instead of using the `return` keyword to provide a result back to the caller, we use the keyword `yield`. When defining a generator, we will almost always include a loop in the body of the function and then we'll `yield` from within the loop.

Here's an example generator:

```
def gen_range(stop, start=1, step=1):
    num = start
    while num <= stop:
        yield num
        num += step
```

When calling a generator function, a generator object is returned instead of the function body being called. To get a value from a generator it is passed into the `next` function:

```
>>> generator = gen_range(4)
>>> next(generator)
1
>>> next(generator)
2
>>> next(generator)
3
>>>
```

This is how a generator works. It loops internally, yielding a result each time it's passed to the `next` function until it reaches the end of the function because it stops looping. Here's what we see if we pass the generator to `next` too many times:

```
>>> next(generator)
4
>>> next(generator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

In practice, we won't normally be calling the `next` function on our generators, we'll be using them with `for` loops like this:

```
>>> for num in gen_range(10, step=2):
...     print(num)
...
1
3
5
7
9
>>>
```

The `for` loop automatically knows how to work with generators so we don't have to worry about running into the `StopIteration` error.

Converting Generators to Lists

If a generator is not an infinite generator then it can be converted into a list by using the built-in `list` function:

```
>>> generator = gen_range(10)
>>> list(generator)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Scopes

In Python, functions (and class) declarations define scopes. This means that any variables created within the function declarations are not available outside of the function. Contexts like conditionals and loops do not define scopes, and variables defined within them are available to the context above. Here's an example of variable scoping:

scopes.py

```
def set_x():
    x = 5

set_x()

while x < 6:
    print(x)
    x += 1

print(x)
```

Now if we run this we'll see the following error because `x` is scoped to the function:

```
$ python3.7 scopes.py
Traceback (most recent call last):
  File "scopes.py", line 7, in <module>
    while x < 6:
NameError: name 'x' is not defined
$
```

Name Hiding (Shadowing)

Name hiding or name shadowing is a concept where we have a variable name in a more specific scope that is the same as a variable at a higher scope. A common example of this is when a function has a parameter that matches a variable that exists at the same level as the function declaration:

```
name = 'Kevin'

def print_name(name):
    print(name)

print_name('Kyle') # Kyle
print(name) # Kevin
```

Because the parameter is at a more specific scope the value of `name` within the function will always be whatever the argument is when the function is called, and it won't change the variable that exists outside of the function.

The `global` Keyword

If we want to have a function interact with global state and variables we can use the `global` keyword:

```
y = 4

def set_y(val):
    global y
    print(y) # accesses global value
    y = val # changes global value

set_y(1) # 4
print(y) # 1
```