Глава 4

Введение в программирование алгоритмов на Python

В ЭТОЙ ГЛАВЕ...

- Числовые и логические вычисления
- Работа со строками
- Работа с датами
- Упаковка кода с помощью функций
- Принятие решений и повторение действий
- Управление данными в памяти
- Чтение данных в объекты хранения
- Быстрый поиск данных с помощью словарей

Любой рецепт приготовления блюда является своего рода алгоритмом, потому что помогает приготовить вкусную еду с помощью ряда действий (и тем самым избавиться от голода). Можно разработать много способов создания последовательности шагов, которые решают задачу. Имеется множество различных процедур и описаний, которые указывают последовательность шагов для решения поставленной задачи. Не всякая последовательность шагов

конкретна. Ряд шагов решения математической задачи может записываться с помощью математических обозначений, но многие люди считают их тайным языком, который мало кто может понять. Компьютерный язык может превратить этот тайный язык в конкретную форму инструкций на языке программирования с использованием английских слов, понятную куда большему количеству людей.

Из главы 3, "Использование Python для работы с алгоритмами", этой книги вы узнали, как установить на компьютер копию Python для работы с примерами из книги. Во всей этой книге Python используется для решения численных задач с использованием алгоритмов, которые можно выразить и с помощью математических формул. Здесь язык программирования используется для того, чтобы превратить эти необычные абстрактные символы в текст, понятный большему количеству людей, которые смогут использовать его для решения своих реальных задач.

Прежде чем можно будет использовать Python для решения задач с помощью алгоритмов, следует по крайней мере узнать о работе Python и с Python. Эта глава не предназначена для того, чтобы сделать из вас эксперта по Python. Однако она предоставит вам достаточно информации, чтобы вы понимали смысл примеров кода с комментариями. Различные разделы помогут понять, как Python выполняет те или иные задачи. Например, знать, как Python работает с различными видами данных, нужно для того, чтобы определить, что именно некоторый пример кода с ними делает. В первых трех разделах вы ознакомитесь с азами работы с числовыми, логическими, строковыми данными и данными, представляющими даты.

Представьте себе поваренную или любую иную книгу, в которой шаги для выполнения различных задач представлены в виде одного длинного рассказа без каких-либо перерывов. Найти в такой книге конкретный рецепт (или описание некоторой процедуры) будет невозможно, и книга будет совершенно бесполезной. На самом деле никто и не попытался бы написать такую книгу. Поэтому в четвертом разделе главы рассматриваются функции, которые сродни отдельным рецептам в поваренной книге. Вы можете комбинировать функции при создании программы, как сочетали бы рецепты различных блюд для приготовления обеда.

В следующих четырех разделах рассматриваются способы управления данными, включая чтение, запись, изменение и удаление. Вам также нужно знать, как принимать решения и выполнять одни и те же наборы действий более одного раза. Данные представляют собой ресурс, так же, как мука, сахар и другие ингредиенты являются ресурсами, которые используются при работе с рецептом. Различные виды данных требуют различных методов работы с ними, чтобы использовать их в приложении, решающем ту или иную задачу,

для которой предназначен рассматриваемый алгоритм. В этих разделах речь идет о способах работы с данными для решения задач.

Работа с числовыми и логическими данными

Работа с алгоритмами включает работу с данными различных видов, но бо́льшая часть работы предусматривает работу с числами. Кроме того, для принятия решений об использовании данных применяются логические значения. Например, может потребоваться узнать, равны ли два числовых значения или одно из них больше другого. Python поддерживает как числовые, так и логические значения.

- Любое число без дробной части является *целочисленным* (integer) значением. Например, таким значением является число `1`. С другой стороны, `1.0` целочисленным не является, так как имеет дробную часть (пусть и нулевую). Целочисленные значения представимы типом данных `int`. На большинстве платформ значение типа `int` может хранить значения от `-9223372036854775808` до `9223372036854775807` (предельные значения, которые могут храниться в 65-битной переменной).
- Любые числа, включающие десятичную дробную часть, являются *значениями с плавающей точкой*. Например, число `1.0` имеет дробную часть, а потому является значением с плавающей точкой. Многие путают целые числа и числа с плавающей точкой, но простейшую разницу между ними легко запомнить: если вы видите в числе десятичную точку, значит, это значение с плавающей точкой. Python хранит значения с плавающей точкой в переменных с типом данных `float`. На большинстве платформ максимальное значение, которое может содержаться в такой переменной, равно $\pm 1.7976931348623157 \times 10^{308}$, а минимальное значение, которое может содержаться в такой переменной, равно $\pm 2.2250738585072014 \times 10^{-308}$.
- *Комплексное число* состоит из пары чисел — действительной и мнимой частей. Если вы совершенно забыли или и вовсе не знали, что такое комплексные числа, можете почитать о них по адресу `http://go.dialektika.com/alg04_01`. Мнимая часть комплексного числа всегда указывается с помощью символа *j* после нее. Так что комплексное число с действительной частью 3 и мнимой частью 4 можно присвоить с помощью выражения наподобие `myComplex=3+4j`.

» Логические аргументы требуют булевых значений, названных так в честь Джорджа Буля (George Bool). Для использования булевых значений в Python имеется тип `bool`. Переменная этого типа может содержать только два значения: `True` и `False`. Можно присвоить булевой переменной значение, используя ключевые слова `True` и `False`, а можно создать выражение, которое определяет истинную или ложную логическую концепцию. Например, можно написать `myBool=1>2`, что равносильно присваиванию значения `False`, потому что 1, определенно, не превышает 2.

Теперь, когда вы познакомились с азами, пришло время увидеть типы данных в действии. Ниже вы найдете краткий обзор о работе с числовыми и логическими данными в Python.

Присваивание переменных

Работая с приложениями, вы храните информацию в *переменных*. Переменная — это своего рода коробка для хранения. Всякий раз, планируя работу с информацией, вы обращаетесь к ней с помощью переменной. Если у вас есть новая информация, которую вы хотите сохранить, вы помещаете ее в переменную. Изменение информации означает доступ к переменной с последующим сохранением нового значения в той же переменной. Так же, как в реальном мире вы храните вещи в коробках, при работе с приложениями вы храните информацию в переменных (своего рода ящиках для хранения). Для сохранения данных в переменной они присваиваются с помощью любого из множества *операторов присваивания* (специальных символов, которые указывают, как следует сохранять данные). В табл. 4.1 показаны поддерживаемые Python операторы присваивания.

Таблица 4.1. Операторы присваивания Python

Оператор	Описание	Пример
`=`	Присваивает значение правого операнда левому	`MyVar = 5` сохраняет в `MyVar` значение 5
`+=`	Добавляет значение правого операнда к значению левого и помещает результат в левый операнд	`MyVar += 2` сохраняет в `MyVar` значение 7
`-=`	Вычитает значение правого операнда из значения левого и помещает результат в левый операнд	`MyVar -= 2` сохраняет в `MyVar` значение 3
`*=`	Умножает значение правого операнда на значение левого и помещает результат в левый операнд	`MyVar *= 2` сохраняет в `MyVar` значение 10

Окончание табл. 4.1

Оператор	Описание	Пример
`/=`	Делит значение левого операнда на значение правого и помещает результат в левый операнд	`MyVar /= 2` сохраняет в `MyVar` значение 2,5
`%=`	Делит значение левого операнда на значение правого и помещает остаток от деления в левый операнд	`MyVar %= 2` сохраняет в `MyVar` значение 1
`**=`	Возводит значение левого операнда в степень, указанную значением правого операнда, и помещает результат в левый операнд	`MyVar **= 2` сохраняет в `MyVar` значение 25
`//=`	Делит значение левого операнда на значение правого и помещает целую часть результата деления в левый операнд	`MyVar //= 2` сохраняет в `MyVar` значение 2

Арифметические действия

Сохранение информации в переменных делает ее легко доступной. Однако, чтобы от информации была какая-то польза, над ней обычно выполняют некоторые действия, чаще всего — арифметические операции. Python поддерживает распространенные арифметические операторы, используемые и при решении задач вручную. Они показаны в табл. 4.2.

Таблица 4.2. Арифметические операторы Python

Оператор	Описание	Пример
`+`	Суммирует два значения	`5 + 2 = 7`
`-`	Вычитает правый операнд из левого	`5 - 2 = 3`
`*`	Умножает правый операнд на левый	`5 * 2 = 10`
`/`	Делит левый операнд на правый	`5 / 2 = 2.5`
`%`	Делит левый операнд на правый и возвращает остаток	`5 % 2 = 1`
`**`	Вычисляет значение левого операнда, возведенное в степень, указываемую значением правого операнда	`5 ** 2 = 25`
`//`	Делит левый операнд на правый нацело, т.е. возвращает только целую часть результата	`5 // 2 = 2`

Иногда требуется выполнять действия с одной переменной — для этого Python поддерживает несколько *унарных операторов*, показанных в табл. 4.3.

Таблица 4.3. Унарные операторы Python

Оператор	Описание	Пример
`~`	Инвертирует биты числа, так что бит 0 становится битом 1, и наоборот	`~4` дает `-5`
`-`	Меняет знак числа, так что положительное значение становится отрицательным, и наоборот	`-(-4)` дает `4`; `-4` дает `-4`
`+`	Предоставляется исключительно для полноты; дает значение, совпадающее с исходным	`+4` дает `4`

Компьютеры могут выполнять и другие виды математических вычислений, связанные с тем, как работает процессор. Важно помнить, что компьютеры хранят данные в виде ряда отдельных битов. Python позволяет получить доступ к этим отдельным битам с помощью побитовых операторов, показанных в табл. 4.4.

Таблица 4.4. Побитовые операторы Python

Оператор	Описание	Пример
`&` (И)	Проверяет, имеют ли соответствующие биты в обоих операндах значения true, и, если имеют, соответствующий бит результата тоже получает значение true, в противном случае — false	`0b1100&0b0110 = 0b0100`
`|` (ИЛИ)	Проверяет, имеют ли соответствующие биты в обоих операндах значения false, и, если имеют, соответствующий бит результата тоже получает значение false, в противном случае — true	`0b1100|0b0110 = 0b1110`
`^` (Исключающее ИЛИ)	Проверяет, имеют ли соответствующие биты в обоих операндах разные значения, и, если имеют, соответствующий бит результата получает значение true, в противном случае — false	`0b1100^0b0110 = 0b1010`

Окончание табл. 4.4

Оператор	Описание	Пример
~ (Дополнение до 1)	Вычисляет значение, являющееся дополнением данного значения до единицы	`~0b1100 = -0b1101` `~0b0110 = -0b0111`
<< (Сдвиг влево)	Сдвигает биты левого операнда влево на количество позиций, указанное правым операндом. Все новые биты — нулевые; биты, выходящие за пределы представления числа, теряются	`0b00110011<<2 = 0b11001100`
>> (Сдвиг вправо)	Сдвигает биты левого операнда вправо на количество позиций, указанное правым операндом. Все новые биты — нулевые; биты, выходящие за пределы представления числа, теряются	`0b00110011>>2 = 0b00001100`

Сравнение данных с помощью булевых выражений

Использование арифметических операций для изменения содержимого переменных является одним из видов манипуляции данными. Чтобы определить результат такой манипуляции, компьютер должен сравнить текущее состояние переменной с ее первоначальным состоянием или с заранее известным значением. В некоторых случаях необходимо сопоставление одних входных данных с другими. Все эти операции проверяют отношения между двумя переменными, поэтому такие операторы (показанные в табл. 4.5) называются реляционными или операторами отношения.

Таблица 4.5. Реляционные операторы Python

Оператор	Описание	Пример
==	Определяет, равны ли два значения. Обратите внимание, что данный оператор использует два знака равенства. Часто допускаемая ошибка состоит в использовании только одного знака равенства, что приводит к присваиванию значения переменной	`1 == 2` равно `False`
!=	Определяет, различаются ли два значения. В некоторых старых версиях Python нужно использовать оператор <> вместо `!=`. В текущей версии применение оператора <> является ошибкой	`1 != 2` равно `True`

Окончание табл. 4.5

Оператор	Описание	Пример
>	Проверяет, является ли значение левого операнда больше значения правого	`1 > 2` равно `False`
<	Проверяет, является ли значение левого операнда меньше значения правого	`1 < 2` равно `True`
>=	Проверяет, является ли значение левого операнда больше или равно значению правого	`1 >= 2` равно `False`
<=	Проверяет, является ли значение левого операнда меньше или равно значению правого	`1 <= 2` равно `True`

Иногда одного реляционного оператора недостаточно для выполнения требуемого сравнения. Например, может потребоваться проверить условие, в котором необходимы два отдельных сравнения, например `MyAge>40` и `MyHeight<174`. Необходимость сравнений из нескольких условий требует применения логических операторов, показанных в табл. 4.6.

Таблица 4.6. Логические операторы Python

Оператор	Описание	Пример

`and`	Определяет, истинны ли одновременно оба операнда	`True and True == True`
		`True and False == False`
		`False and True == False`
		`False and False == False`

`or`	Определяет, истинен ли хотя бы один операнд	`True or True == True`
		`True or False == True`
		`False or True == True`
		`False or False == False`

`not`	Обращает истинность операнда — значение `True` становится значением `False`, и наоборот	`not True == False` `not False = True`

Компьютеры упорядочивают сравнения, делая одни операторы более важными, чем другие. Упорядочение операторов определяется *приоритетами операторов*. В табл. 4.7 показаны приоритеты всех распространенных операторов Python, включая и такие, с которыми вы еще не сталкивались. При выполнении сравнений всегда учитывайте приоритет операторов, потому что в противном

случае предположения, которые вы делаете о результатах сравнения, скорее всего, будут неправильными.

Таблица 4.7. Приоритеты операторов в Python

Оператор	Описание
`()`	Используйте скобки для группирования выражений и переопределения приоритета по умолчанию. Так вы можете заставить операцию с более низким приоритетом (например, сложение) иметь приоритет перед операцией с более высоким приоритетом (например, умножением)
`**`	Возведение левого операнда в степень, определяемую правым операндом
`~ + -`	Унарные операторы взаимодействуют с единственным значением или выражением
`* / % //`	Умножение, деление, остаток при делении, деление нацело
`+ -`	Сложение и вычитание
`>> <<`	Побитовые сдвиги вправо и влево
`&`	Побитовое И
`^ |`	Побитовые исключающее и стандартное ИЛИ
`<= < > >=`	Операторы сравнения
`== !=`	Операторы равенства
`= %= /= //= -= += *= **=`	Операторы присваивания
`is` `is not`	Операторы тождества
`in` `not in`	Операторы членства
`not or and`	Логические операторы

Создание и использование строк

Среди всех типов данных строки являются наиболее понятными для людей (и непонятными для компьютеров). *Строка* — это просто группа символов, заключенная в двойные кавычки. Например, выражение `myString="Python is a great language."` выполняет присваивание строки переменной `myString`.

ЗАПОМНИ!

Основная причина использования строк при работе с алгоритмами — обеспечение взаимодействия с пользователем, например, для запросов входных данных или для облегчения понимания выводимых данных. В рамках работы с алгоритмами можно также выполнить анализ строковых данных, но фактически компьютер не требует использования строк в качестве части последовательности шагов для решения задачи. На самом деле компьютер вообще не видит букв. Каждая используемая буква представляет собой число в памяти. Например, буква `A` — это в действительности число 65. Чтобы убедиться в этом, введите **`ord("A")`** в приглашении Python и нажмите клавишу <Enter>. Вы увидите выведенное число 65. Вы можете преобразовать любую букву в ее числовой эквивалент, используя команду `ord()`.

РАБОТА С IPYTHON

Большая часть книги основана на Jupyter Notebook (см. главу 3, "Использование Python для работы с алгоритмами"), потому что он предоставляет методы для создания, управления и взаимодействия со сложными примерами. Однако иногда требуется простая интерактивная среда для быстрых тестов. Anaconda поставляется с двумя такими средами, IPython и Jupyter QT Console. Хотя обе среды предоставляют схожие функциональные возможности, IPython является более простой в использовании. Чтобы запустить IPython, просто щелкните на ее пиктограмме в папке Anaconda3 своей системы. Работая в Windows, можете выбрать команду Start⇨All Programs⇨Anaconda3⇨IPython (Пуск⇨Все программы⇨Anaconda3⇨IPython в русскоязычной версии). Вы также можете запустить IPython в консольном окне или терминале, введя **`IPython`** и нажав клавишу <Enter>.

Поскольку компьютер в действительности не понимает строки, которые так полезны при написании приложений, иногда требуется преобразовать строку в число. Для выполнения такого преобразования можно использовать команды `int()` и `float()`. Например, если в приглашении Python ввести **`myInt = int("123")`** и нажать клавишу <Enter>, можно создать переменную типа `int` с именем `myInt`, содержащую значение `123`.

ЗАПОМНИ!

Можно преобразовывать и числа в строку, используя команду `str()`. Например, если ввести **`myStr = str(1234.56)`** и нажать клавишу <Enter>, будет создана строка, содержащая значение `"1234.56"`, и присвоена переменной `myStr`. Идея в том, что вы очень легко можете

выполнять преобразования чисел в строки и обратно. В последующих главах будет показано, как эти преобразования делают множество, казалось бы, невозможных задач вполне выполнимыми.

Как и с числами, со строками (и многими другими объектами) можно использовать некоторые специальные операторы. *Операторы членства* (member operators) позволяют определить, когда в строке находится некоторое определенное содержимое. Эти операторы показаны в табл. 4.8.

Таблица 4.8. Операторы членства Python

Оператор	Описание	Пример
`in`	Определяет, имеется ли значение левого операнда в последовательности, находящейся в правом операнде	`"Hello" in "Hello Goodbye"` равно `True`
`not in`	Определяет, отсутствует ли значение левого операнда в последовательности, находящейся в правом операнде	`"Hello" not in "Hello Goodbye"` равно `False`

Обсуждение в этом разделе также делает очевидной необходимость знания типа данных, содержащихся в переменных. Для выполнения этой задачи используются *операторы тождественности*, показанные в табл. 4.9.

Таблица 4.9. Операторы тождественности Python

Оператор	Описание	Пример
`is`	Возвращает `True`, если тип значения или выражения правого операнда совпадает с типом левого операнда	`type(2) is int` равно `True`
`is not`	Возвращает `True`, если тип значения или выражения правого операнда отличается от типа левого операнда	`type(2) is not int` равно `False`

Работа с датами

Дата и время знакомы большинству людей и постоянно ими используются. В нашем обществе почти все основано на датах и времени, и, в первую очередь,

к ним привязано решение тех или иных задач. Мы строим планы, определяя конкретные дату и время, когда будем заниматься той или иной работой. Мы живем по часам, обедая, отдыхая или развлекаясь в точно определенные часы. Весь наш день вращается вокруг часов. При работе с алгоритмами дата или время, когда выполняется некоторый шаг последовательности, могут быть такими же важными, как и сам шаг и то, что происходит в результате его выполнения. Алгоритмы часто используют дату и время для организации данных, чтобы люди могли лучше понимать как данные, так и результат работы алгоритма.

В силу сказанного нам следует взглянуть на то, как Python работает с датой и временем (в частности, как он хранит соответствующие значения для последующего использования). Как и только что рассмотренные строки, компьютеры не понимают дату и время, которые в действительности для них не существуют, — они могут работать только с числами. С датой и временем, по сути, работает не компьютер, а алгоритм.


ЗАПОМНИ!

Чтобы работать с датами и временем, необходимо использовать специальную команду — `import datetime`. Технически это действие называется *импортом модуля*. Пока что не волнуйтесь о том, как работает эта команда, — просто используйте ее всякий раз, когда хотите сделать что-то с датой или временем.

В компьютеры встроены часы, но они предназначены для людей, которые пользуются ими с помощью компьютера. Конечно, некоторое программное обеспечение также зависит от часов, но все же основной акцент делается на потребности человека, а не компьютера. Чтобы получить текущее время, можно просто ввести команду **`datetime.datetime.now()`** и нажать клавишу <Enter>. Вы увидите полную информацию о текущих дате и времени, как на часах компьютера, например `datetime.datetime(2016,12,20,10,37,24,460099)`.

Вероятно, вы обратили внимание на трудность восприятия показанного формата даты и времени. Пусть, например, вы хотите получить только текущую дату в удобном для восприятия формате. Для выполнения этой задачи вы обращаетесь только к той части выходных данных, которая относится к дате, и преобразуете ее в строку. Введите **`str(datetime.datetime.now().date())`** и нажмите клавишу <Enter>. Теперь вы получите немного более понятный вывод, такой как `'2017-07-06'`.

Достаточно интересно, что в Python также есть команда `time()`, которую можно использовать для получения текущего времени. Вы можете получить отдельные значения каждого компонента даты и времени, используя значения `day`, `month`, `year`, `hour`, `minute`, `second` и `microsecond`. Последующие главы помогут вам понять, как использовать различные функции даты и времени для упрощения работы с алгоритмами.

Создание и использование функций

Для каждого шага алгоритма обычно требуется одна строка кода Python — инструкция, которая указывает компьютеру, как на один шаг приблизиться к полному решению задачи. Вы объединяете эти строки кода для достижения желаемого результата. Иногда требуется повторить инструкцию с различными данными, а в некоторых случаях код становится настолько длинным, что становится трудно следить за тем, что делает та или иная его часть. Функции служат инструментом организации кода, который позволяет поддерживать чистоту и опрятность ваших исходных текстов. Кроме того, функции облегчают повторное использование кода при необходимости работы с различными данными. Этот раздел данной главы полностью посвящен функциям. Здесь вы, кроме того, начнете создавать свои первые серьезные приложения так же, как это делают профессиональные разработчики.

Создание повторно используемых функций

Вы идете к шкафу, достаете из него брюки и рубашку, снимаете этикетки и одеваетесь. В конце дня вы снимаете их и выбрасываете в мусорное ведро. Мда… Не слишком похоже на правду. Большинство людей снимает одежду, стирает и кладет обратно в шкаф для повторного использования. Функции тоже являются многоразовыми. Никому не понравится детально расписывать одни и те же действия — это быстро становится монотонным и скучным делом. При создании функции определяется пакет кода, который можно повторно использовать для выполнения одной той же задачи — любое количество раз. Все, что вам нужно сделать, — это сообщить компьютеру, какую именно функцию использовать. Компьютеру никогда не надоедает добросовестно выполнять все инструкции функции всякий раз, когда вы просите его это сделать.

ЗАПОМНИ!

При работе с функциями код, который нуждается в услугах функции, называется *вызывающим*; он вызывает функцию для выполнения некоторых действий. Вызывающий код должен предоставить функции необходимую для ее работы информацию; функция, в свою очередь, также возвращает информацию вызывающему коду.

В свое время компьютерные программы не включали концепцию повторного использования кода. В результате разработчикам приходилось снова и снова изобретать один и тот же код. Довольно быстро была придумана идея функций; эта концепция прошла достаточно долгий путь, пока функции не стали такими гибкими, какими они являются сегодня. Вы можете сделать функцию выполняющей любые нужные вам действия. Повторное использование кода является необходимой частью приложений для

- ускорения разработки;
- уменьшения количества ошибок при программировании;
- увеличения надежности приложений;
- использования группой результатов работы одного программиста;
- облегчения понимания кода;
- повышения эффективности приложения.

Фактически функции выполняют целый список действий в виде многократного использования в приложениях. Выполняя примеры в данной книге, вы увидите, как повторное использование упрощает жизнь. Если бы не повторное использование кода, вы по-прежнему программировали бы, вручную вводя нули и единицы в компьютер.

Для создания функции не требуется много работы. Чтобы увидеть, как работают функции, откройте копию IPython и введите следующий код (нажимая клавишу <Enter> в конце каждой строки):

```
def SayHello():
    print('Hello There!')
```

Для завершения функции нажмите <Enter> еще раз после последней строки. Функция начинается с ключевого слова `def` (“define” — “определение”). Далее указываются имя функции, скобки, которые могут содержать *аргументы* функции (данные, передаваемые в функцию), и двоеточие. В следующей строке редактор автоматически делает отступ. Python использует пробелы для определения *блоков кода* (инструкций, которые в функции связаны одна с другой).

Теперь вы можете использовать функцию. Просто введите **`SayHello()`** и нажмите клавишу <Enter>. Скобки после имени функции обязательно должны быть в наличии, поскольку они говорят Python о том, что вы ожидаете выполнения функции, а не обращаетесь к ней как к объекту. В результате работы этой функции будет выведена строка `Hello There!`.

Вызовы функций

Функции могут принимать аргументы и возвращать значения. Возможность обмена данными делает функции гораздо более полезными, чем они могли бы быть в противном случае. В следующих разделах описано, как вызывать функции, чтобы обеспечить возможность обмена данными.

Передача аргументов в функцию

Функция может потребовать от вызывающего кода предоставить аргументы. Обязательный аргумент — это переменная, которая должна содержать данные, необходимые для работы функции. Откройте копию IPython и введите следующий код:

```
def DoSum(Value1, Value2):
    return Value1 + Value2
```

Теперь у вас есть новая функция, `DoSum()`. Она требует, чтобы при ее использовании вы предоставляли ей два аргумента. Если вы введете **`DoSum()`** (без аргументов) и нажмете клавишу <Enter>, то увидите сообщение об ошибке наподобие следующего:

```
TypeError
    Traceback (most recent call last)
<ipython-input-2-a37c1b30cd89> in <module>()
----> 1 DoSum()

TypeError: DoSum() missing 2 required positional
    arguments: 'Value1' and 'Value2'
```

Попытка вызвать `DoSum()` только с одним аргументом приведет к другому сообщению об ошибке. Корректное использование `DoSum()` требует предоставления ей двух аргументов. Чтобы увидеть, как это работает, введите **`DoSum(1,2)`** и нажмите клавишу <Enter>. Вы увидите ожидаемый результат — `3`.


ЗАПОМНИ!

Обратите внимание, что функция `DoSum()` предоставляет выходное значение `3` при передаче ей в качестве входных данных `1` и `2`. Это выходное значение возвращает оператор `return`. Увидев в функции оператор `return`, вы знаете, что эта функция возвращает некоторое выходное значение.

Передача аргументов с помощью ключевых слов

По мере того как ваши функции становятся все более и более сложными, как и методы их использования, может потребоваться обеспечить несколько больший контроль над вызовом функции и передаче ей аргументов. До сих пор мы использовали *позиционные аргументы*. Это означает, что значения предоставлялись в том порядке, в котором они появляются в списке аргументов в определении функции. Однако Python имеет и другой метод передачи аргументов — с помощью ключевых слов. В этом случае необходимо указать имя аргумента, за которым следуют знак равенства (=) и значение аргумента. Чтобы увидеть, как это работает, откройте копию IPython и введите следующий код:

```
def DisplaySum(Value1, Value2):
    print(str(Value1) + ' + ' + str(Value2) + ' = ' +
    str((Value1 + Value2)))
```


ЗАПОМНИ!

Обратите внимание, что аргумент функции `print()` включает список элементов для вывода и что эти элементы разделены знаком “плюс” (+). Кроме того, аргументы имеют различные типы, поэтому вначале необходимо преобразовать их с помощью функции `str()`. Python

позволяет легко комбинировать аргументы таким способом. Кроме того, данная функция вводит концепцию автоматического продолжения строки. Функция `print()` фактически располагается в двух строках, и Python автоматически продолжает чтение кода функции, переходя из первой строки во вторую.

Теперь самое время протестировать функцию `DisplaySum()`. Сначала проверим, как она работает с использованием позиционных аргументов. Для этого введите **`DisplaySum(2,3)`** и нажмите клавишу <Enter>. Вы увидите ожидаемый результат: `2 + 3 = 5`. Теперь введите **`DisplaySum(Value2=3,Value1=2)`** и нажмите клавишу <Enter>. Вы вновь получите вывод `2 + 3 = 5`, несмотря на то что аргументы в функцию теперь переданы в обратном порядке.

Использование аргументов со значениями по умолчанию

Вызываете ли вы функцию, используя ключевое слово или позиционные аргументы, в любом случае вы должны указать значения аргументов. Но иногда функция может использовать значения аргументов по умолчанию, если известно некоторое часто используемое значение. Значения по умолчанию делают функцию проще в использовании и снижают вероятность ошибки, когда разработчик не предоставляет для функции входные данные. Для создания значения по умолчанию в определении функции достаточно после имени аргумента добавить знак равенства и значение по умолчанию. Чтобы увидеть, как это работает, откройте копию IPython и введите следующий код:

```
def SayHello(Greeting = "No Value Supplied"):
    print(Greeting)
```

Функция `SayHello()` обеспечивает значение приветствия по умолчанию, если вызывающий код не предоставляет соответствующий аргумент. Когда кто-то пытается вызвать `SayHello()` без аргументов, это не приводит к ошибке. Введите **`SayHello()`** и нажмите клавишу <Enter>, чтобы увидеть сообщение по умолчанию. Если вы введете, например, **`SayHello("Howdy!")`**, то увидите вывод переданной в функцию строки.

Создание функций с переменным числом аргументов

В большинстве случаев вы точно знаете, какое количество аргументов должна получать ваша функция. Желательно использовать в программах именно такие функции, поскольку функции с фиксированным числом аргументов упрощают поиск и устранение проблем позже. Однако иногда вы просто не можете заранее определить, сколько аргументов будет получать ваша функция. Например, при создании приложения Python, работающего в командной строке, пользователь может не предоставить ему ни одного аргумента, максимальное

количество аргументов (в предположении, что такое значение существует) или любое промежуточное количество аргументов.

К счастью, Python предоставляет возможность передачи переменного количества аргументов функции. Для этого вы просто создаете аргумент, перед именем которого имеется звездочка, например `*VarArgs`. Обычно при этом функции передается еще один аргумент, который содержит количество аргументов, передаваемых функции в качестве входных данных. Чтобы увидеть, как это работает, откройте копию IPython и введите следующий код:

```
def DisplayMulti(ArgCount = 0, *VarArgs):
    print('Вы передали ' + str(ArgCount) + ' аргумента.', VarArgs)
```

Обратите внимание, что функция `print()` выводит строку, а затем список аргументов. Благодаря дизайну этой функции можно просто ввести **`DisplayMulti()`** и нажать клавишу <Enter> — так вы увидите, что этой функции можно передать нулевое количество аргументов. Чтобы просмотреть, как работает вызов с несколькими аргументами, введите **`DisplayMulti(3, 'Hello',1,True)`** и нажмите клавишу <Enter>. Полученный вывод `('Вы передали 3 аргумента.', ('Hello', 1, True))` показывает, что при этом в функцию можно передавать значения любого типа.

Использование условных и циклических инструкций

Алгоритмы часто содержат шаги, в которых принимаются те или иные решения или которые выполняются более одного раза. Например, может потребоваться отбросить значение, которое не соответствует остальным входным данным, что требует принятия соответствующего решения, или для получения желаемого результата может потребоваться обработать данные более одного раза, например при фильтрации данных. Python удовлетворяет эти потребности путем предоставления описанных в следующих разделах специальных инструкций, которые позволяют принимать решения или выполнять шаги более одного раза.

Принятие решений с помощью инструкций `if`

В повседневной жизни вы регулярно используете предложения со словом “если”. Например, вы можете сказать себе “если сегодня четверг, на обед я возьму блюдо из рыбы”. Инструкции `if` в Python менее многословны, но действуют по той же схеме. Чтобы увидеть, как это работает, откройте копию IPython и введите следующий код:

```
def TestValue(Value):
    if Value == 5:
        print('Значение Value равно 5!')
    elif Value == 6:
        print('Значение Value равно 6!')
    else:
        print('Значение Value равно чему-то иному.')
        print('Оно равно ' + str(Value))
```

Каждая инструкция `if` в Python начинается, как ни странно это звучит, со слова `if`. Когда Python видит `if`, он знает, что вы хотите принять некоторое решение. После слова `if` идет *условие*, указывающее, какой вид сравнения должен выполнить Python. В приведенном выше случае вы хотите, чтобы Python выяснил, содержит ли `Value` значение `5`.

ВНИМАНИЕ!

Обратите внимание, что условие использует оператор отношения равенства ==, а не оператор присваивания =. Распространенная ошибка начинающих разработчиков — использование оператора присваивания вместо оператора равенства, что ведет к некорректной работе кода.

Условие всегда заканчивается двоеточием (`:`). Если вы не поставите двоеточие, Python не будет знать, что условие закончено, и будет продолжать искать дополнительные условия, что, скорее всего, приведет к ошибке. После двоеточия следуют действия, выполнения которых вы хотите от Python.

Может потребоваться выполнить несколько задач в одной инструкции `if`. Конструкция `elif` позволяет добавить дополнительное условие и связанные с ним задачи. Эта *конструкция* представляет собой дополнение к предыдущему условию, которым в данном случае является конструкция `if`. Конструкция `elif` всегда предусматривает наличие условия, как в инструкции `if`, и имеет собственный связанный с условием набор задач для выполнения.

Иногда вам нужно сделать что-то, если условие не выполняется. Для этого служит инструкция `else`. Она указывает Python, что делать, если условия в конструкции `if` оказываются ложными.

ЗАПОМНИ!

Обратите внимание, как с усложнением функциональности растет важность отступов. Функция содержит инструкцию `if`, которая, в свою очередь, содержит только один вызов `print()`. Конструкция `else` содержит уже два вызова `print()`.

Чтобы увидеть эту функцию в действии, введите **`TestValue(1)`** и нажмите клавишу <Enter>. Вы увидите выход из конструкции `else`. Введите **`TestValue(5)`** и нажмите клавишу <Enter>, и вывод будет выполнен вызовом из инструкции `if`. Если вы введете **`TestValue(6)`** и нажмете клавишу <Enter>, то выполнится вызов из конструкции `elif`. Функция `TestValue()` оказывается

значительно более гибкой, чем предыдущие функции, рассматриваемые в этой главе, потому что она может принимать решения.

Выбор одного из нескольких вариантов с использованием вложенных принятий решений

Вложенность представляет собой процесс размещения подчиненной инструкции внутри другой. В большинстве случаев вы можете вложить любую инструкцию в любую другую. Чтобы увидеть, как это работает, откройте копию IPython и введите следующий код:

```
def SecretNumber():
    One = int(input("Введите число от 1 до 10: "))
    Two = int(input("Введите число от 1 до 10: "))
    if (One >= 1) and (One <= 10):
        if (Two >= 1) and (Two <= 10):
            print('Ваше секретное число равно: ' + str(One * Two))
        else:
            print("Неверное второе значение!")
    else:
        print("Неверное первое значение!")
```

В этом случае функция `SecretNumber()` просит вас ввести два числа. При необходимости вы можете получить данные от пользователя с помощью функции `input()`; функция `int()` преобразует введенные данные в число.

На этот раз имеется два уровня инструкции `if`. Первый уровень проверяет корректность числа `One`. Второй уровень проверяет корректность числа `Two`. Когда и `One`, и `Two` имеют значения от 1 до 10, функция `SecretNumber()` выводит пользователю "секретное число".

Чтобы увидеть функцию `SecretNumber()` в действии, введите **`SecretNumber()`** и нажмите клавишу <Enter>. Введите **`20`** и нажмите клавишу <Enter>, когда вас попросят ввести первое входное значение, а в ответ на просьбу ввести второе значение введите **`10`** и нажмите клавишу <Enter>. Вы увидите сообщение об ошибке, гласящее, что первое значение введено неверно. Введите **`SecretNumber()`** и нажмите клавишу <Enter> еще раз. На этот раз введите значения **`10`** и **`20`**. Теперь функция сообщит, что второе значение является неверным. Попробуйте еще раз выполнить те же действия, вводя на этот раз числа **`10`** и **`10`**.

Выполнение повторяющихся действий с использованием цикла `for`

Иногда необходимо выполнить некоторые действия более чем один раз. Когда нужно выполнить действия определенное количество раз, можно

использовать цикл for, который имеет определенные начало и конец. Количество итераций, выполняемых циклом, зависит от количества элементов в предоставляемой переменной. Чтобы увидеть, как это работает, откройте копию IPython и введите следующий код:

```
def DisplayMulti(*VarArgs):
    for Arg in VarArgs:
        if Arg.upper() == 'CONT':
            continue
            print('Continue Argument: ' + Arg)
        elif Arg.upper() == 'BREAK':
            break
            print('Break Argument: ' + Arg)
        print('Good Argument: ' + Arg)
```

В этом случае цикл for пытается обработать каждый элемент VarArgs. Обратите внимание на наличие вложенной в цикл инструкции if, которая проверяет два условия завершения.

В большинстве случаев код пропускает инструкцию if и просто выводит аргумент. Однако, когда инструкция if находит во входных значениях слово CONT или BREAK, она выполняет одно из двух действий.

- continue: заставляет цикл продолжаться, переходя от текущей точки к обработке следующего значения из VarArgs.
- break: прекращает работу цикла.

СОВЕТ

Ключевые слова могут использовать любую комбинацию прописных и строчных букв, как, например, ConT, потому что функция upper() преобразует их в символы верхнего регистра. Функция DisplayMulti() может обрабатывать любое количество входных строк. Чтобы увидеть ее в действии, введите **DisplayMulti('Hello','Goodbye','First','Last')** и нажмите клавишу <Enter>. Вы увидите каждую из входных строк в отдельной строке на экране в выходных данных. Теперь введите **DisplayMulti('Hello','Cont','Goodbye','Break','Last')** и нажмите клавишу <Enter>. Обратите внимание, что слова Cont и Break в выходных данных не отображаются, потому что являются ключевыми словами. Кроме того, в выходных данных отсутствует и слово Last, поскольку цикл завершается до его обработки.

Использование цикла while

Цикл while выполняет итерации до тех пор, пока его условие не перестанет быть истинным. Как и инструкция for, инструкция while поддерживает

ключевые слова `continue` и `break`. Чтобы увидеть цикл `while` в работе, откройте копию IPython и введите следующий код:

```
def SecretNumber():
    GotIt = False
    while GotIt == False:
        One = int(input("Введите число от 1 до 10: "))
        Two = int(input("Введите число от 1 до 10: "))

        if (One >= 1) and (One <= 10):
            if (Two >= 1) and (Two <= 10):
                print('Ваше секретное число равно: ' + str(One * Two))
                GotIt = True
                continue
            else:
                print("Неверное второе значение!")
        else:
            print("Неверное первое значение!")
        print("Попробуйте еще раз!")
```

Перед вами расширение функции `SecretNumber()`, описанной в разделе "Выбор одного из нескольких вариантов с использованием вложенных принятий решений" ранее в этой главе. Однако в этом случае добавление инструкции `while` приводит к тому, что функция продолжает запрашивать повтор ввода, пока не получит корректный ответ.

Чтобы увидеть, как работает цикл `while`, введите **`SecretNumber()`** и нажмите клавишу <Enter>. Введите **`20`** и нажмите клавишу <Enter>, затем введите **`10`** и вновь нажмите клавишу <Enter>. Вы получите сообщение о том, что первое введенное число неверное, и предложение повторить ввод. Попробуйте во второй раз, используя теперь значения **`10`** и **`20`**. На этот раз вы получите сообщение о неверном втором числе и предложение повторить ввод. При третьей попытке используйте значения **`10`** и **`10`**. На этот раз вы получите "секретное число". Обратите внимание, что оператор `continue` не позволяет приложению предложить вам снова ввести данные.

Хранение данных в множествах, списках и кортежах

При работе с алгоритмами все действия выполняются с данными. Python предоставляет множество методов для хранения данных в памяти. Каждый метод имеет свои преимущества и недостатки. Важно, исходя из конкретных потребностей, выбрать наиболее подходящий метод хранения. В следующих разделах рассматриваются три распространенных метода хранения данных.

Создание множеств

Большинство людей использовали множества (да хотя бы на уроках математики в школе) для создания списков связанных между собой элементов. Затем над этими списками могут выполняться различные манипуляции с использованием математических операций, таких как пересечение, объединение, разность или симметричная разность. Множества оказываются наилучшим выбором, когда нужно выполнить проверку членства или удалить дубликаты из списка. С помощью множеств нельзя выполнить задачи, связанные с последовательностями, такие как индексирование или срез. Чтобы на практике увидеть, как можно работать с множествами, запустите копию IPython и введите следующий код:

```
SetA = set(['Red', 'Blue', 'Green', 'Black'])
SetB = set(['Black', 'Green', 'Yellow', 'Orange'])
SetX = SetA.union(SetB)
SetY = SetA.intersection(SetB)
SetZ = SetA.difference(SetB)
```

Теперь у вас есть пять различных множеств, каждое из которых имеет некоторые общие элементы. Чтобы просмотреть результаты каждой математической операции, введите **`print('{0}\n{1}\n{2}'.format(SetX,SetY,SetZ))`** и нажмите клавишу <Enter>. Вы увидите по одному множеству в каждой строке:

```
{'Blue', 'Orange', 'Red', 'Green', 'Black', 'Yellow'}
{'Green', 'Black'}
{'Blue', 'Red'}
```

СОВЕТ

Вывод демонстрирует результаты выполнения математических операций `difference()`, `union()` и `intersection()`. Форматирование вывода в Python может оказаться полезным при работе с множествами. Функция `format()` указывает Python, какие объекты следует использовать вместо каждого из заполнителей в строке. *Заполнитель* представляет собой фигурные скобки (`{}`) с необязательным числом в них. *Управляющий символ* `\n` обеспечивает вывод знака начала новой строки. О применении форматирования и управлении им в Python вы можете прочесть по адресу `http://go.dialektika.com/alg04_02`.

Можно также проверить различные соотношения между множествами. Например, введите **`SetA.issuperset(SetY)`** и нажмите клавишу <Enter>. Выходное значение `True` говорит о том, что множество `SetA` является надмножеством `SetY`. Аналогично, введя **`SetA.issubset(SetX)`** и нажав клавишу <Enter>, вы обнаружите, что `SetA` представляет собой подмножество `SetX`.

Важно понимать, что множества могут быть изменяемыми или неизменяемыми. В данном примере все множества являются изменяемыми, т.е. в них можно добавлять элементы или удалять их из множеств. Например, если ввести **`SetA.add('Purple')`** и нажать клавишу <Enter>, множество `SetA` получит новый элемент. Если вы введете **`SetA.issubset(SetX)`** и нажмете клавишу <Enter>, то выяснится, что `SetA` больше не является подмножеством `SetX`, потому что у множества `SetA` имеется элемент `'Purple'`, которого нет в `SetX`.

Создание списков

Список в Python определен как разновидность последовательности. *Последовательность* представляет собой средство, позволяющее нескольким элементам данных сосуществовать в одной единице хранения, но в качестве отдельных сущностей. Рассматривайте их как ящики для почты в подъездах многоквартирных домов. Один такой ящик содержит ряд небольших ячеек, каждая из которых может содержать почту. Python также поддерживает другие виды последовательностей.

- **Кортежи**. Кортеж представляет собой коллекцию, которая используется для создания сложных, похожих на список последовательностей. Преимуществом кортежей является то, что содержимое кортежа может быть вложенным. Эта функциональность позволяет создавать структуры, которые могут содержать записи о сотрудниках или о парах координат `x-y`.
- **Словари**. При использовании словарей вы создаете пары "ключ/значение" (как и в случае реальных словарей пары "слово/его определение"). Словарь обеспечивает невероятно быстрый поиск и упрощает упорядочение данных.
- **Стеки**. Большинство языков программирования непосредственно поддерживает стеки. Однако в Python стек не поддерживается, хотя для этого имеется обходной путь. Стек — это последовательность, которая характеризуется правилом "последним вошел — первым вышел" (last in/first out — LIFO). Подумайте о стопке блинов: вы можете добавлять новые блины, укладывая их поверх стопки, но и брать их можно тоже только сверху. Стек представляет собой важную коллекцию, которую можно моделировать в Python с помощью списка.
- **Очереди**. Очередь представляет собой коллекцию, которая характеризуется правилом "первым вошел — первым вышел" (first in/first out — FIFO). Очередь используется для отслеживания элементов, которые должны быть каким-то образом обработаны. Думайте об очереди как об обычной очереди в реальной жизни.

» **Деки**. Двусторонняя очередь (double-ended queue, deque) представляет собой структуру, схожую с очередью, но у нее постановка в очередь и выход из нее могут осуществляться с обоих концов (но не из средины). Дек можно использовать как очередь или как стек, или как любую иную коллекцию, добавление в которую (и удаление из которой) выполняется упорядоченным образом (в отличие от списков, кортежей и словарей, которые допускают произвольный доступ к элементам).

Среди всех последовательностей списки являются простейшими для понимания и наиболее тесно связанными с реальными объектами. Работа со списками помогает научиться работе с другими видами последовательностей, обеспечивающими большую функциональность и гибкость. Дело в том, что хранение данных в списке очень схоже с их записью на листке бумаги, когда один элемент следует за другим. Список имеет начало, средину и конец. Python нумерует элементы в списке. (Нумерация элементов в списке облегчает доступ к ним.) Чтобы просмотреть списки в работе, запустите копию IPython и введите следующий код:

```
ListA = [0, 1, 2, 3]
ListB = [4, 5, 6, 7]
ListA.extend(ListB)
ListA
```

После ввода последней строки кода вы увидите вывод `[0,1,2,3,4,5,6,7]`. Функция `extend()` добавляет элементы-члены списка `ListB` к списку `ListA`. Помимо расширения списков, к ним можно добавлять элементы, используя функцию `append()`. Введите **`ListA.append(-5)`** и нажмите клавишу <Enter>. Если вы введете **`ListA`** и снова нажмете <Enter>, то увидите, что Python добавил `-5` в конец списка. Чтобы удалить элементы из списка, используйте функцию `remove()`. Например, введите **`ListA.remove(-5)`** и нажмите клавишу <Enter>. Когда вы снова запросите список `ListA`, введя **`ListA`** и нажав <Enter>, вы увидите, что добавленный элемент `-5` из списка удален.


ЗАПОМНИ!

Списки также поддерживают *конкатенацию* с помощью оператора “плюс” (+), который добавляет элементы одного списка к другому. Например, если ввести **`ListX=ListA+ListB`** и нажать клавишу <Enter>, можно увидеть, что созданный список `ListX` содержит элементы списков `ListA` и `ListB`, причем элементы списка `ListA` расположены первыми.

Создание и использование кортежей

Кортеж представляет собой коллекцию, которая используется для создания сложных списков, в которых один кортеж можно вставлять в другой. Это позволяет создавать иерархии с использованием кортежей. Иерархия может быть по уровню сложности такой же, как список каталогов жесткого диска или организационная структура вашей компании. Главное, что вы можете создавать сложные структуры данных с помощью кортежей.

ЗАПОМНИ!

Кортежи являются *неизменяемыми*, так что вы не можете их изменять. Можно создать новый кортеж с тем же именем и тем или иным способом его изменить, но нельзя изменить существующий кортеж. В отличие от кортежей, списки являются изменяемыми. На первый взгляд, неизменяемость кортежей может показаться недостатком, но она имеет свои преимущества, а именно — безопасность и скорость. Кроме того, неизменяемые объекты легче использовать в многозадачной среде с несколькими процессорами. Чтобы увидеть, как работают кортежи, запустите копию IPython и введите следующий код:

```
MyTuple = (1, 2, 3, (4, 5, 6, (7, 8, 9)))
```

MyTuple имеет трехуровневую вложенность. Первый уровень состоит из значений `1`, `2`, `3` и кортежа. Второй уровень состоит из значения `4`, `5`, `6` и еще одного кортежа. Третий уровень состоит из значений `7`, `8` и `9`. Чтобы увидеть, как это работает, введите в IPython следующий код:

```
for Value1 in MyTuple:
    if type(Value1) == int:
        print(Value1)
    else:
        for Value2 in Value1:
            if type(Value2) == int:
                print("\t", Value2)
            else:
                for Value3 in Value2:
                    print("\t\t", Value3)
```

При выполнении этого кода вы увидите значения на трех разных уровнях, показанных с помощью отступов:

```
1
2
3
    4
    5
    6
        7
        8
        9
```

СОВЕТ

Можно выполнять такие задачи, как добавление новых значений, но делать это следует путем добавления в новый кортеж исходных записей и новых значений. Кроме того, кортежи можно добавлять только в уже существующие кортежи. Чтобы увидеть, как это работает, введите **`MyNewTuple=MyTuple.__add__((10,11,12,(13,14,15)))`** и нажмите клавишу <Enter>. Кортеж `MyNewTuple` теперь содержит новые записи на первом и втором уровнях: `(1,2,3,(4,5,6,(7,8,9)),10,11,12,(13,14,15))`.

Определение итераторов

В последующих главах используются всевозможные методы доступа к отдельным значениям в структурах данных различных типов. В этом разделе используются два простых списка, определенных следующим образом:

```
ListA = ['Orange', 'Yellow', 'Green', 'Brown']
ListB = [1, 2, 3, 4]
```

Простейший метод получения доступа к определенным значениям состоит в использовании индекса. Например, если ввести **`ListA[1]`** и нажать клавишу <Enter>, на выходе можно увидеть `'Yellow'`. Все индексы в Python отсчитываются с нуля, а это означает, что первая запись имеет индекс 0, а не 1.

Диапазоны представляют собой еще одно простое средство доступа к значениям. Например, если ввести **`ListB[1:3]`** и нажать <Enter>, будет выведено `[2,3]`. Диапазон можно использовать в качестве входных данных для цикла `for`, такого как

```
for Value in ListB[1:3]:
    print(Value)
```

Вместо всего списка вы увидите выведенными в отдельных строках только `2` и `3`. Диапазон состоит из двух значений, разделенных двоеточием. Однако значения являются необязательными. Например, `ListB[:3]` приведет к выводу `[1,2,3]`. Когда вы пропускаете значение, диапазон начинается в начале (или заканчивается в конце) списка.

Иногда нужно обработать два списка параллельно. Самый простой способ сделать это — использовать функцию `zip()`. Ниже приведен пример функции `zip()` в действии:

```
for Value1, Value2 in zip(ListA, ListB):
    print(Value1, '\t', Value2)
```

Этот код одновременно обрабатывает `ListA` и `ListB`. Обработка завершается, когда цикл `for` достигает конца более короткого из двух списков. В этом случае можно увидеть следующее:

```
Orange 1
Yellow 2
Green 3
Brown 4
```

ЗАПОМНИ!

Это всего лишь верхушка айсберга. В книге вы увидите множество различных типов итераторов. Идея заключается в том, чтобы позволить перечислять только нужные элементы вместо всех элементов в списке или иной структуре данных. Некоторые из итераторов, используемых в последующих главах, немного сложнее показанных здесь, но это всего лишь начало.

Индексация данных с использованием словарей

Словарь представляет собой особый вид последовательности, которая использует пару "имя/значение". Используя имя, можно легко получить доступ к определенному значению с помощью сущности, отличной от числового индекса. Для создания словаря пары имен и значений заключаются в фигурные скобки. Создайте тестовый словарь, введя **`MyDict={'Orange':1,'Blue':2, 'Pink':3}`** и нажав клавишу <Enter>.

Для доступа к определенному значению используйте в качестве индекса имя. Например, введите **`MyDict['Pink']`** и нажмите <Enter>, чтобы увидеть выходное значение `3`. Использование словарей в качестве структуры данных упрощает доступ к невероятно сложным наборам данных с использованием понятных для всех терминов. Во многих других отношениях работа со словарем является такой же, как и с любой другой последовательностью.

Словари обладают некоторыми специальными возможностями. Например, можно ввести **`MyDict.keys()`** и нажать клавишу <Enter>, чтобы просмотреть список ключей. Чтобы увидеть список значений в словаре, можно воспользоваться функцией `values()`.