Практики

Тема 1. Введение в Python. Основные конструкции и базовые типы.
Выбор среды разработки (IDE). Базовые типы. Условные операторы. Циклы. Файлы.
Модули и пакеты. Виртуальное окружение (Virtualenv).
Установка и запуск Jupyter Notebook.

Выбор среды разработки (IDE)

Писать код можно в любом текстовом редакторе, однако специализированные редакторы и полноценные среды разработки (IDE) добавляют процессу программирования массу удобств – таких как подсветка синтаксиса, автодополнение, интроспекция кода (возможность по клику перейти к месту объявления используемого класса или функции) и многие другие.

Если вы не уверены, в чем вам программировать – попробуйте PyCharm Community Edition. Это полноценная IDE, которая позволит вам даже запускать код на Python, не выходя из редактора. Чтобы установить PyCharm Community Edition, перейдите по ссылке – вам будет предложено скачать на выбор Professional или Community версию среды разработки PyCharm. Professional версия – платная, для полноценной работы с Python будет достаточно бесплатной Community версии. Скачайте установщик PyCharm Community Edition для вашей операционной системы. После этого запустите установщик и следуйте инструкциям.

Также можете посмотреть в сторону Visual Studio Code – для разработки на Python вам дополнительно потребуется установить плагин "Python".

Если вы программист с опытом, то возможно вам будет удобно писать код на Python в редакторах Vim или Emacs.

Среди других вариантов можно отметить редактор Atom, а также любимый многими за простоту и расширяемость Sublime Text.

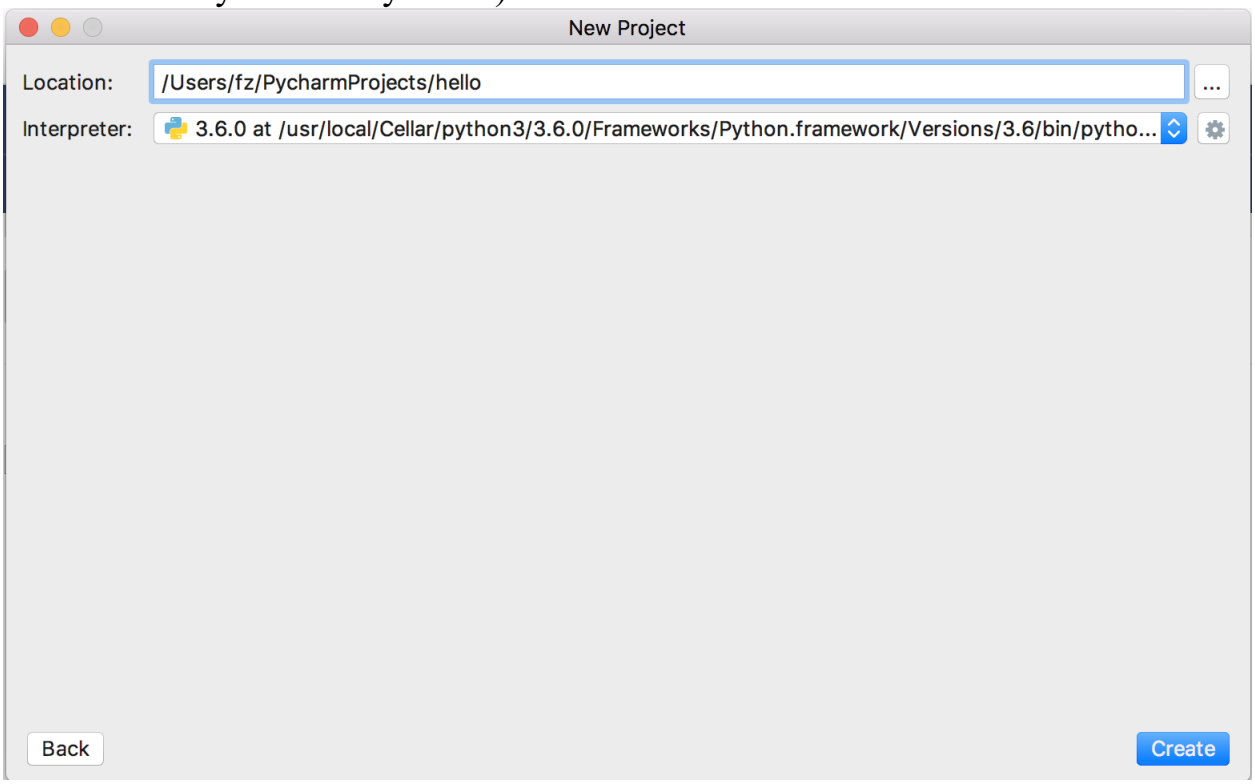
Как вы можете видеть – редакторов и IDE много, выбирайте по вкусу. Мы же подробно опишем процесс создания нового Python проекта в PyCharm Community Edition – если не знаете какой редактор выбрать, советуем остановиться именно на нем.

Создание нового проекта в PyCharm Community Edition

Когда PyCharm открылся нажмите Create New Project (Создать Новый Проект).

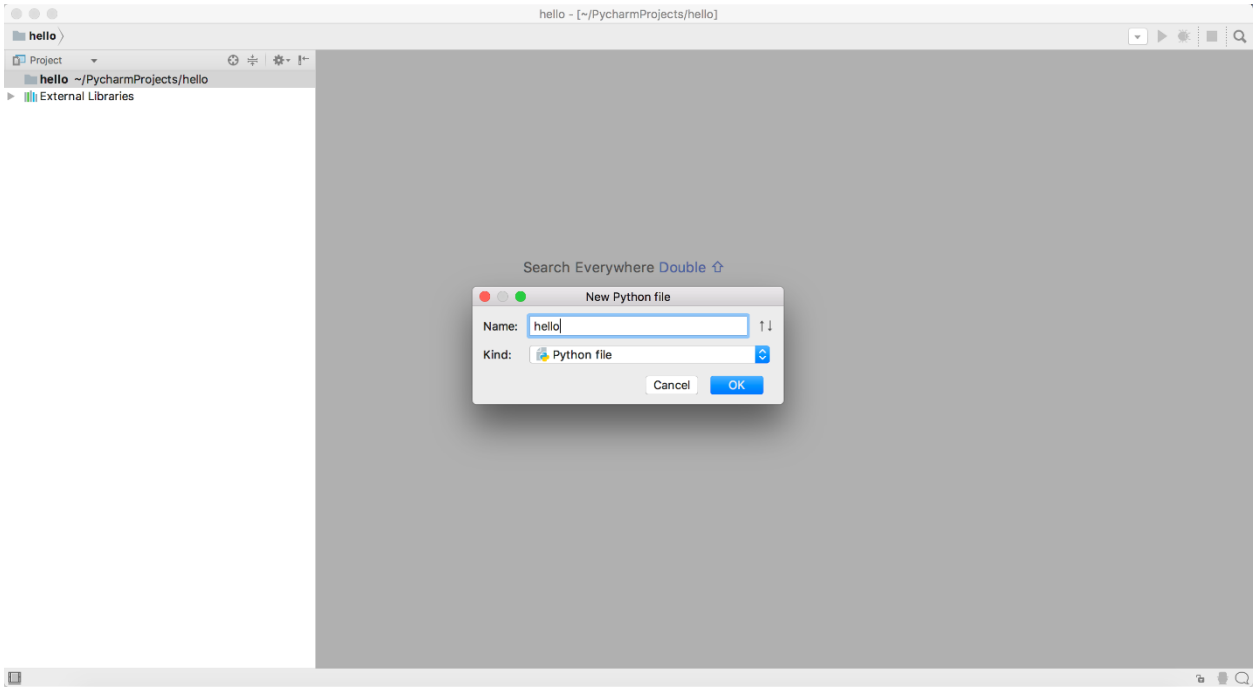
Задайте путь и имя проекта, а также путь до интерпретатора Python 3, который вы установили. Если вы не знаете, где на файловой системе находится путь до интерпретатора, – вот как можно его найти:

- На Windows в терминале наберите **where python3** (или **where python** – в зависимости от того, как у вас запускается Python 3)
- На Unix-системах (Linux, MacOS ...) наберите в терминале **which python3** (или **which python** – в зависимости от того, как у вас запускается Python 3)



После задания имени проекта и пути до интерпретатора в окне PyCharm нажмите кнопку Create.

Откроется окно редактора. Слева на боковой панели нажмите на директории с именем вашего проекта правой кнопкой мыши и выберите New -> Python File.



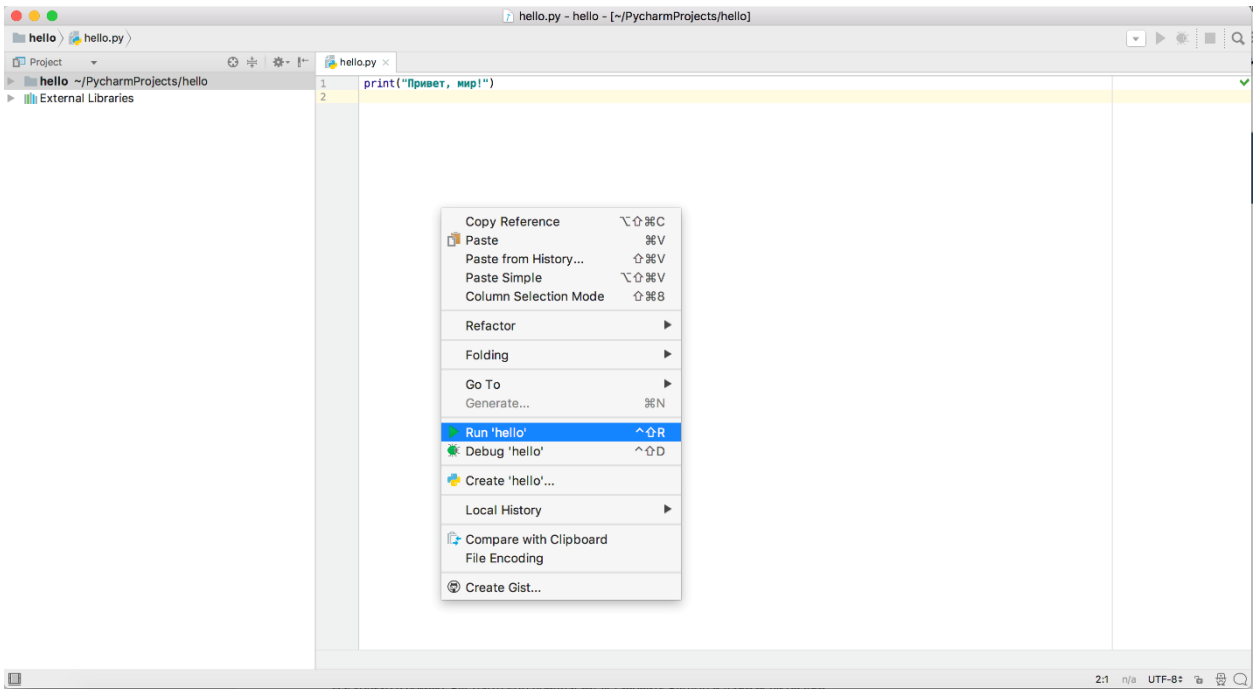
Введите имя файла (например, hello) и нажмите OK – PyCharm создаст файл и автоматически добавит ему расширение .py. Созданный файлик откроется в окне редактора – можно начинать писать код на Python.

Напишите туда:

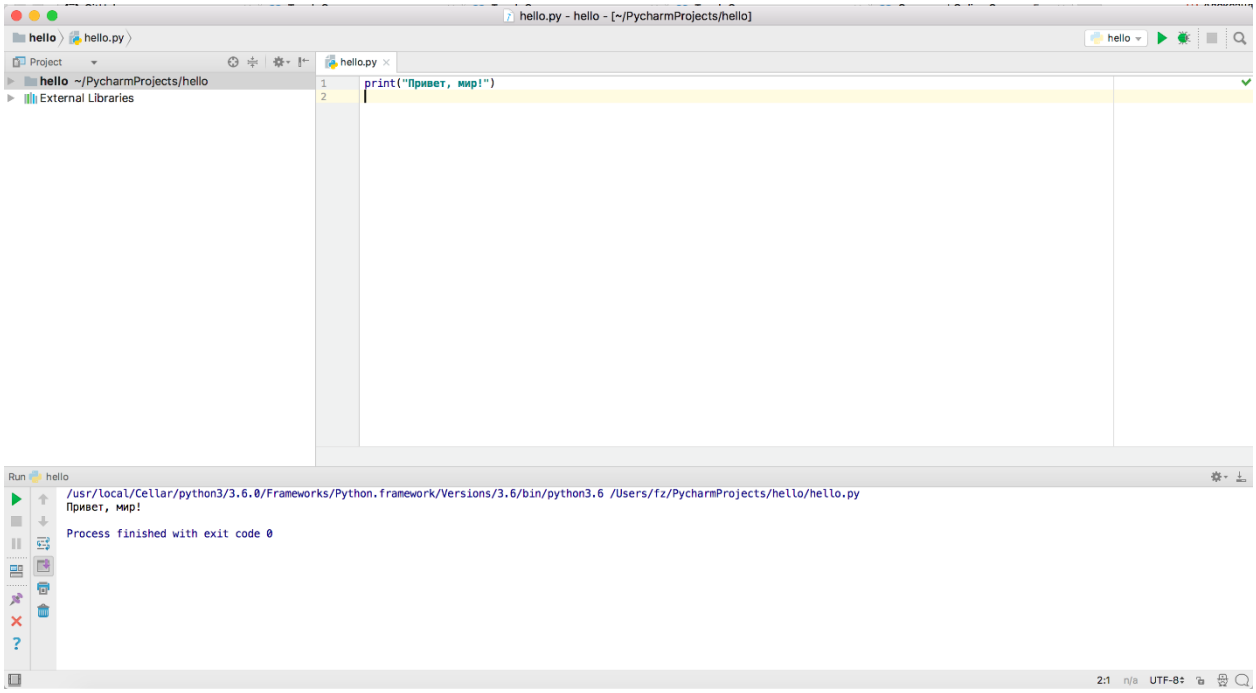
```
1
```

```
print("Привет, мир!")
```

А затем щелкните правой кнопкой мыши на окно ввода кода и нажмите Run "hello".



Внизу должно появиться окно терминала, и в нем должен присутствовать вывод нашей программы.



Немного о типизации языков программирования

Если достаточно формально подходить к вопросу о типизации языка Python, то можно сказать, что он относится к языкам с неявной сильной динамической типизацией.

Неявная типизация означает, что при объявлении переменной вам не нужно указывать её тип, при явной – это делать необходимо. В качестве примера языков с явной типизацией можно привести Java, C++. Вот как будет выглядеть объявление целочисленной переменной в Java и Python.

Java:

```
int a = 1;
```

Python:

```
a = 1
```

Также языки бывают с динамической и статической типизацией. В первом случае тип переменной определяется непосредственно при выполнении программы, во втором – на этапе компиляции (о компиляции и интерпретации кратко рассказано в уроке 2). Как уже было сказано Python – это динамически типизированный язык, такие языки как C, C#, Java – статически типизированные.

Сильная типизация не позволяет производить операции в выражениях с данными различных типов, слабая – позволяет. В языках с сильной типизацией вы не можете складывать например строки и числа, нужно все приводить к одному типу. К первой группе можно отнести Python, Java, ко второй – C и C++.

За более подробным разъяснением данного вопроса советуем обратиться к статье “Ликбез по типизации в языках программирования”.

Типы данных в Python

В Python типы данных можно разделить на встроенные в интерпретатор *(built-in*) и не встроенные, которые можно использовать при импортировании соответствующих модулей.

К основным встроенным типам относятся:

1. *None* (неопределенное значение переменной)
2. Логические переменные (*Boolean Type*)
3. Числа (*Numeric Type*)
    1. *int* – целое число
    2. *float* – число с плавающей точкой
    3. *complex* – комплексное число
4. Списки (*Sequence Type*)
    1. *list* – список
    2. *tuple* – кортеж
    3. *range* – диапазон
5. Строки (*Text Sequence Type*)
    1. *str*
6. Бинарные списки (*Binary Sequence Types*)
    1. *bytes* – байты
    2. *bytearray* – массивы байт
    3. *memoryview* – специальные объекты для доступа к внутренним данным объекта через protocol buffer
7. Множества (*Set Types*)
    1. *set* – множество
    2. *frozenset* – неизменяемое множество
8. Словари (*Mapping Types*)
    1. *dict* – словарь

При изучении программирования в качестве практики часто приходится создавать «идеальные программы», которые в реальном мире работают совсем не так. Иногда, например, нужно исполнить ряд инструкций только в том случае, если соблюдаются определенные условия. Для обработки таких ситуаций в языках программирования есть операторы управления. В дополнение к управлению потоком выполнения программы эти операторы используются для создания циклов или пропуска инструкций, когда какое-то условие истинно.

Операторы управления бывают следующих типов:

1. Оператор-выражение if
2. Оператор-выражение if-else
3. Оператор-выражение if-elif-else
4. **Цикл while**
5. **Цикл for**
6. Оператор-выражение break
7. Оператор-выражение continue

Оператор if

Синтаксис оператора if следующий:

```
if condition:
    <indented statement 1>
    <indented statement 2>

<non-indented statement>
```

Первая строчка оператора, то есть `if condition`: — это условие if, а `condition` — это логическое выражение, которое возвращает `True` или `False`. В следующей строке блок инструкций. Блок представляет собой одну или больше инструкций. Если он идет следом за условием if, такой блок называют блоком if.

Стоит обратить внимание, что у каждой инструкции в блоке if одинаковый отступ от слова if. Многие языки, такие как C, C++, Java и PHP, используют фигурные скобки (`{}`), чтобы определять начало и конец блока, но в Python используются отступы.

Каждая инструкция должна содержать одинаковое количество пробелов. В противном случае программа вернет синтаксическую ошибку. В документации Python рекомендуется делать отступ на 4 пробела. Такая рекомендация актуальная для и для этого .

Как это работает:

Когда выполняется инструкция if, проверяется условие. Если условие истинно, тогда все инструкции в блоке if выполняются. Но если условие оказывается неверным, тогда все инструкции внутри этого блока пропускаются.

Инструкции следом за условием if, у которых нет отступов, не относятся к блоку if. Например, <non-intenden statement> — это не часть блока if, поэтому она будет выполнена в любом случае.

Например:

```
number = int(input("Введите число: "))



if number > 10:

    print("Число больше 10")
```

Первый вывод:

Введите число: 100

Число больше 10

Второй вывод:

Введите число: 5

Стоит обратить внимание, что во втором случае, когда условие не истинно, инструкция внутри блока if пропускается. В этом примере блок if состоит из одной инструкции, но их может быть сколько угодно, главное — делать отступы.

Рассмотрим следующий код:

```
number = int(input("Введите число: "))
if number > 10:
    print("первая строка")
    print("вторая строка")
    print("третья строка")


print("Выполняется каждый раз, когда вы запускаете программу")
print("Конец")
```

Первый вывод:

```
Введите число: 45
первая строка
вторая строка
третья строка
Выполняется каждый раз, когда вы запускаете программу
Конец
```

Второй вывод:

```
Введите число: 4
Выполняется каждый раз, когда вы запускаете программу
Конец
```

Здесь важно обратить внимание, что только выражения на строках 3, 4 и 5 относятся к блоку if. Следовательно, они будут исполнены только в том случае, когда условие if будет истинно. Но инструкции на строках 7 и 8 выполнятся в любом случае.

Консоль Python реагирует иначе при использовании операторов управления прямо в ней. Стоит напомнить, что для разбития выражения на несколько строк используется оператор продолжение (\). Но в этом нет необходимости с операторами управления. Интерпретатор Python автоматически активирует мультистрочный режим, если нажать Enter после условия if. Например:

```
>>>

>>> n = 100

>>> if n > 10:

...
```

После нажатия Enter на строке с условием if командная строка преобразуется с >>> на …. Консоль Python показывает … для многострочных инструкций. Это значит, что начатая инструкция все еще не закончена.

Чтобы закончить инструкцию if, нужно добавить еще одну инструкцию в блок if:

```
>>>

>>> n = 100

>>> if n > 10:

...     print("n v 10")

...
```

Python не будет автоматически добавлять отступ. Это нужно сделать самостоятельно. Закончив ввод инструкции, нужно дважды нажать Enter, чтобы исполнить инструкцию. После этого консоль вернется к изначальному состоянию.

```
>>>

>>> n = 100

>>> if n > 10:

...     print("n больше чем 10")

...
```

n больше чем 10

>>>

Все эти программы заканчиваются внезапно, не показывая ничего, если условие не истинно. Но в большинстве случаев пользователю нужно показать хотя бы что-нибудь. Для этого используется оператор-выражение if-else.

Вложенные операторы if и if-else

Использовать операторы if-else можно внутри других инструкций if или if-else. Это лучше объяснить на примерах:

Оператор if внутри другого if-оператора

Пример 1: программа, проверяющая, имеет ли студент право на кредит.

```
gre_score = int(input("Введите текущий лимит: "))

per_grad = int(input("Введите кредитный рейтинг: "))



if per_grad > 70:

    # внешний блок if

        if gre_score > 150:

            # внутренний блок if

                print("Поздравляем, вам выдан кредит")

else:

    print("Извините, вы не имеете права на кредит")
```

Здесь оператор if используется внутри другого if-оператора. Внутренним называют вложенный оператором if. В этом случае внутренний оператор if относится к внешнему блоку if, а у внутреннего блока if есть только одна инструкция, которая выводит “Поздравляем, вам выдан кредит”.

Как это работает:

Сначала оценивается внешнее условие if, то есть per_grad > 70. Если оно возвращает True, тогда управление программой происходит внутри внешнего блока if. Там же проверяется условие gre_score > 150. Если оно возвращает True, тогда в консоль выводится "Congratulations you are eligible for loan". Если False, тогда программа выходит из инструкции if-else, чтобы исполнить следующие операции. Ничего при этом не выводится в консоль.

При этом если внешнее условие возвращает False, тогда выполнение инструкций внутри блока if пропускается, и контроль переходит к блоку else (9 строчка).
Первый вывод:
Введите текущий лимит: 160
Введите кредитный рейтинг: 75
Поздравляем, вам выдан кредит
Второй вывод:
Введите текущий лимит: 160
Введите кредитный рейтинг: 60
Извините, вы не имеете права на кредит
У этой программы есть одна маленькая проблема. Запустите ее заново и введите gre_score меньше чем 150, а per_grade — больше 70:
Введите кредитный рейтинг: 80
Программа не выводит ничего. Причина в том, что у вложенного оператора if нет условия else. Добавим его в следующем примере.
Пример 2: инструкция if-else внутри другого оператора if.

```
gre_score = int(input("Введите текущий лимит: "))
per_grad = int(input("Введите кредитный рейтинг: "))

if per_grad > 70:
    if gre_score > 150:
        print("Поздравляем, вам выдан кредит")
    else:
        print("У вас низкий кредитный лимит")
else:
    print("Извините, вы не имеете права на кредит")
```

Вывод:
Введите текущий лимит: 140
Введите кредитный рейтинг: 80
У вас низкий кредитный лимит
Как это работает:
Эта программа работает та же, как и предыдущая. Единственное отличие — у вложенного оператора if теперь есть инструкция else. Теперь если ввести балл GRE меньше, чем 150, программа выведет: “У вас низкий кредитный лимит”
При создании вложенных операторов if или if-else, всегда важно помнить об отступах. В противном случае выйдет синтаксическая ошибка.

Оператор if-elif-else

Оператор f-elif-else — это альтернативное представление оператора if-else, которое позволяет проверять несколько условий, вместо того чтобы писать вложенные if-else. Синтаксис этого оператора следующий:

```
if condition_1:
    # блок if
    statement
    statement
    more statement
elif condition_2:
    # первый блок elif
    statement
    statement
    more statement
elif condition_3:
    # второй блок elif
    statement
    statement
    more statement

...

else
    statement
    statement
    more statement
```

Примечание: … означает, что можно писать сколько угодно условий eilf.

Как это работает:

Когда исполняется инструкция if-elif-else, в первую очередь проверяется condition_1. Если условие истинно, тогда исполняется блок инструкций if. Следующие условия и инструкции пропускаются, и управление переходит к операторам вне if-elif-else.

Если condition_1 оказывается ложным, тогда управление переходит к следующему условию elif, и проверяется condition_2. Если оно истинно, тогда исполняются инструкции внутри первого блока elif. Последующие инструкции внутри этого блока пропускаются.

Этот процесс повторяется, пока не находится условие elif, которое оказывается истинным. Если такого нет, тогда исполняется блок else в самом конце.

Перепишем программу с помощью if-elif-else.

```
score = int(input("Введите вашу оценку: "))

if score >= 90:
    print("Отлично! Ваша оценка A")
elif score >= 80:
```

```
    print("Здорово! Ваша оценка - B")
elif score >= 70:
    print("Хорошо! Ваша оценка - C")
elif score >= 60:
    print("Ваша оценка - D. Стоит повторить материал.")
else:
    print("Вы не сдали экзамен")
```

Первый вывод:
Введите вашу оценку: 78
Хорошо! Ваша оценка - C
Второй вывод:
Введите вашу оценку: 91
Отлично! Ваша оценка A
Третий вывод:
Введите вашу оценку: 55
Вы не сдали экзамен.
Такую программу намного легче читать, чем в случае с вложенными if-else.

Виртуальное окружение (Virtualenv). Установка и запуск Jupyter Notebook
Чтобы поставить сторонний пакет на операционную систему, пользуются утилитой pip. В командной строке можно набрать pip install и название сторонней библиотеки, которую вам хочется поставить. Хорошей практикой при программировании на Python является использование виртуальных окружений. Виртуальное окружение в Python --- это окружение, которое позволяет изолировать зависимости для определённого проекта. Обычно для каждого проекта создаётся своё виртуальное окружение, в которое ставятся все пакеты, используемые в данном проекте. Чтобы создать виртуальное окружение, воспользуемся модулем venv. Синтаксис: python3 −m venv и далее название директории, в которой будет создано новое виртуальное окружение. Обычно папка с виртуальным окружением создаётся рядом с папкой проекта. Виртуальное окружение активируется следующей командой: source /bin/activate После этого в виртуальное окружение можно устанавливать пакеты известной нам командой pip install. Например, поставим известное приложение Jupyter Notebook. Jupyter Notebook --- очень распространённое приложение, которое применяется повсеместно разработчиками на Python, а также учёными для презентации своей работы и интерактивного запуска кода. pip install jupyter Вместе с jupyter установится полезный пакет ipython --- расширенная версия интерактивного интерпретатора Python, которая может служить его альтернативой. Чтобы запустить Jupyter Notebook, воспользуйтесь командой jupyter-notebook. Это приложение запускается в веб-браузере. Внутри него можно создавать "ноутбуки", где в ячейках можно писать код и интерактивно его исполнять.

Тема 2. Структуры данных. Коллекции.
Списки и кортежи. Словари. Множества.
Коллекция --- это переменная-контейнер, в которой может содержаться какое-то количество объектов, где объекты могут быть одного типа или разного. В случае списков это упорядоченные наборы элементов, которые могут быть разных типов. Сами списки определяются с помощью квадратных скобочек или с помощью вызова литерала list. Вы также можете создать список из одинаковых значений с помощью умножения. Несмотря на вышесказанное, чаще всего списки содержат переменные одного типа. Также списки могут содержать другие коллекции, как, например, user_data. Однако для таких данных чаще всего используются кортежи, о которых будет сказано позже. empty_list = [] empty_list = list() none_list = [None] * 10 collections = ['list', 'tuple', 'dict', 'set']

Словари являются важнейшей структурой данных в Python-е. Они позволяют хранить данные в формате ключ-значение. Чтобы определить словарь, нужно использовать литерал фигурные скобки или просто вызвать dict. Если мы хотим, определяя словарь, сразу добавить в него данные, пишем ключ-значение через двоеточие. empty_dict = {} empty_dict = dict() collections_map = { 'mutable': ['list', 'set', 'dict'], 'immutable': ['tuple', 'frozenset'] }

Множество в питоне — это неупорядоченный набор уникальных объектов. Множества изменяемы и чаще всего используются для удаления дубликатов и всевозможных проверок на вхождение. Чтобы объявить пустое множество, можно воспользоваться литералом set или использовать фигурные скобки, чтобы объявить множество и одновременно добавить туда какие-то элементы. empty_set = set() number_set = {1, 2, 3, 3, 4, 5} print(number_set) {1, 2, 3, 4, 5}

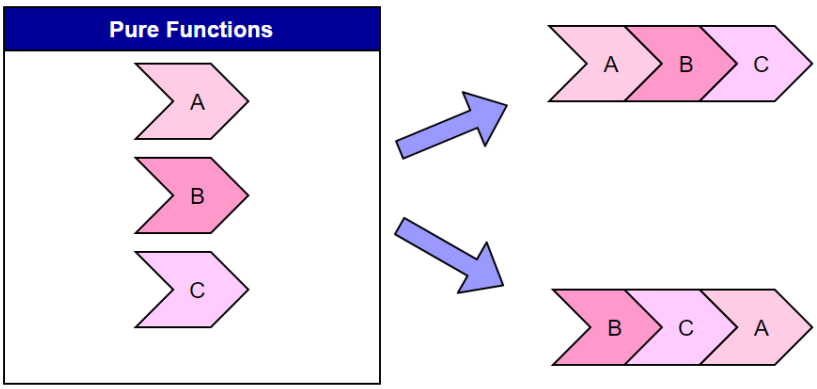
Тема 3. Функциональное программирование. Функции.
map, filter, reduce, partial, lambda — анонимные функции. Списочные выражения
Декораторы. Генераторы.
Функция --- это блок кода, который можно переиспользовать несколько раз в разных местах программы. Мы можем передавать функции аргументы и получать возвращаемые значения. Чтобы определить функцию в языке Python, нужно использовать литерал def и с помощью отступа определить блок кода функции.

Функциональное программирование — это парадигма декларативного программирования, в которой программы создаются путем последовательного применения функций, а не инструкций.

Каждая из этих функций принимает входное значение и возвращает согласующееся с ним выходное значение, не изменяясь и не подвергаясь воздействию со стороны состояния программы.

Для таких функций предусмотрено выполнение только одной операции, если же требуется реализовать сложный процесс, то используется уже композиция функций, связанных последовательно. В процессе ФП мы создаем код, состоящий из множества модулей, поскольку функции в нем могут повторно использоваться в разных частях программы путем вызова, передачи в качестве параметров или возвращения.

Чистые функции не производят побочных эффектов и не зависят от глобальных переменных или состояний.



Визуальное представление функций в ФП

Функциональное программирование используется, когда решения легко выражаются с помощью функций и не имеют ощутимой связи с физическим миром. В то время как объектно-ориентированные программы моделируют код по образцу реальных объектов, ФП задействует математические функции, в которых промежуточные или конечные значения не сопоставляются с объектами физического мира.

К наиболее распространенным областям, применяющим ФП, относятся проектирование ИИ, алгоритмы классификации в МО, финансовые программы, а также продвинутые модели математических функций.

Проще говоря: функциональные программы выполняют много чистых однозадачных функций, совмещенных в последовательность для решения сложных математических или не связанных с физическим миром задач.

Преимущества функционального программирования

- **Легкая отладка**: чистые функции и неизменяемые данные упрощают обнаружение мест определения значений переменных. В чистых функциях меньше факторов, влияющих на них, что позволяет быстрее находить проблемные участки кода.

- **Отложенное вычисление**: функциональные программы производят вычисления только при необходимости. Это позволяет им повторно использовать ранее полученные результаты и экономить время на выполнение.

- **Модульность**: чистые функции не полагаются на внешние переменные или состояния, в связи с чем их можно легко переиспользовать в разных местах программы. Кроме того, функции будут выполнять только одну операцию или вычисление, что не позволит вам при их использовании случайно импортировать лишний код.

- **Лучшая читаемость**: функциональные программы легко читать, потому что поведение каждой функции неизменяемо и изолировано от состояния программы. В результате вы зачастую можете легко понять, что будет делать функция, просто по ее имени.

- **Параллельное программирование**: программы легче создавать при помощи функционального подхода, потому что неизменяемые переменные снижают число изменений внутри этих программ. Каждой функции приходится работать только с вводом пользователя, и она может быть уверена, что состояние программы в основном останется прежним.

Языки функционального программирования

Функциональная парадигма поддерживается не во всех языках. Некоторые из них, например Haskell, спроектированы именно для этой задачи, в то время как другие, например JavaScript, реализуют возможности и ООП, и ФП. Есть же и такие языки, где функциональное программирование невозможно в принципе.

Переменные и функции

Ключевыми составляющими функциональной программы являются уже не объекты и методы, а переменные и функции. При этом следует избегать глобальных переменных, потому что изменяемые глобальные переменные усложняют понимание программы и ведут к появлению у функций побочных эффектов.

Чистые функции

Для чистых функций характерны два свойства:

- они не создают побочных эффектов;
- они всегда производят одинаковый вывод при получении одинакового ввода, что еще можно называть как ссылочную прозрачность.

Побочные эффекты же возникают, если функция изменяет состояние программы, переписывает вводную переменную или в общем вносит какие-либо изменения при генерации вывода. Отсутствие же побочных эффектов снижает риски появления ошибок по вине чистых функций.

Ссылочная прозрачность означает, что любой вывод функции должен допускать замену на ее значение, не изменяя при этом результата программы. Этот принцип гарантирует, что вы создаете такие функции, которые выполняют только одну операцию и достигают согласованного вывода.

Ссылочная прозрачность возможна только, если функция не влияет на состояние программы или в общем не старается выполнить более одной операции.

Неизменяемость и состояния

Неизменяемые данные или состояния не могут изменяться после их определения, что позволяет сохранять постоянство стабильной среды для вывода функций. Лучше всего программировать каждую функцию так, чтобы она выводила один и тот же результат независимо от состояния программы. Если же она зависит от состояния, то это состояние должно быть неизменяемым, чтобы вывод такой функции оставался постоянным.

Подходы функционального программирования обычно избегают применения функций с общим состоянием (когда несколько функций опираются на одно состояние) и функций с изменяющимся состоянием (которые зависят от изменяемых функций), потому что они уменьшают модульность программы. Если же вы не можете обойтись без функций с общим состоянием, сделайте это состояние неизменяемым.

Рекурсия

Одно из серьезных отличий объектно-ориентированного программирования от функционального в том, что программы последнего избегают таких конструкций, как инструкции if else или циклы, которые в разных случаях выполнения могут выдавать разные выводы.

Вместо циклов функциональные программы используют для всех задач по перебору рекурсию.

Функции первого класса

Функции в ФП рассматриваются как типы данных и могут использоваться как любое другое значение. Например, мы заполняем функциями массивы, передаем их в качестве параметров или сохраняем их в переменных.

Функции высшего порядка

Эти функции могут принимать другие функции в качестве параметров или возвращать функции в качестве вывода. Они делают возможности вызова функций более гибкими и позволяют легче абстрагироваться от действий.

Композиция функций

Для выполнения сложных операций функции можно выполнять последовательно. В этом случае результат каждой функции передается следующей функции в виде аргумента. Это позволяет с помощью всего одного вызова функции активировать целую серию их последовательных вызовов.

Функциональное программирование в Python

В Python реализована частичная поддержка ФП, и некоторые используемые в нем решения математических программ легче реализуются с помощью именно функционального подхода.

Самая сложная часть перехода к использованию такого подхода в сокращении числа используемых классов. В Python классы имеют изменяемые атрибуты, что усложняет создание чистых неизменяемых функций.

Попробуйте оформлять весь код на уровне модулей и переключайтесь на классы только по мере необходимости.

Давайте посмотрим, как добиться чистых неизменяемых функций и функций первого класса в Python, после чего познакомимся с синтаксисом для их композиции.

Чистые и неизменяемые функции

Многие из встроенных в Python структур данных являются неизменяемыми по умолчанию:

- integer;
- float;
- Boolean;
- string;
- Unicode;
- tuple.

Кортежи особенно полезны при использовании в качестве неизменяемой формы массива.

```
# код Python для проверки неизменяемости кортежейtuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```

Этот код вызывает ошибку, потому что старается переопределить неизменяемый объект кортежа. Эти неизменяемые структуры данных рекомендуется использовать в функциональных программах Python для получения чистых функций.

Нижеприведенную функцию можно считать чистой, так как у нее нет побочных эффектов, и она всегда возвращает одинаковый вывод:

```
def add_1(x):
   return x + 1
```

Функции первого класса

Отметим, что в Python функции рассматриваются как объекты, и ниже мы приводим краткое руководство по их возможному использованию:

Функции в качестве объектов

```
def shout(text):
   return text.upper()
```

Передача функции в качестве параметра

```
def shout(text):
   return text.upper()def greet(func):
   # сохраняем функцию в переменной
   greeting = func("Hi, I am created by a function passed as an argument.")
   print greeting  greet(shout)
```

Возвращение функции из другой функции

```
def create_adder(x):
   def adder(y):
      return x+y     return adder
```

Композиция функций

Для компоновки функций в Python мы используем вызов lambda function. Это позволяет нам единовременно вызывать любое число аргументов.

```
import functoolsdef compose(*functions):
   def compose2(f, g):
      return lambda x: f(g(x))
   return functools.reduce(compose2, functions, lambda x: x)
```

На **строке 4** мы определяем функцию compose2, получающую две функции в качестве аргументов f и g.
На **строке 5** мы возвращаем новую функцию, представляющую композицию из f и g.

В завершении на **строке 6** мы возвращаем результаты этой композиции функций.

Тема 4. Классы и объекты. Наследование в Python.

Классы и экземпляры. Методы. Наследование. Классы исключений. Объектно-ориентированное программирование --- это особый способ организации кода. Часто говорят, что классы используют тогда, когда нужно отобразить реальные предметы на программный код. Отчасти это так, но в общем случае классы служат для объединения функционала, связанного общей идеей и смыслом, в одну сущность, у которой может быть свое внутреннее состояние, а также методы, которые позволяют модифицировать это состояние. Реальный пример класса: обертка над соединением к базе

данных (состояние --- постоянное TCP-соединение с базой, методы класса предоставляют интерфейс доступа к соединению). Тем самым TCP соединение инкапсулируется внутри класса, а пользователю класса предоставляем удобный интерфейс доступа к данным. Много примеров классов можно найти в реализации игр жанра RPG. Квесты, монстры, игроки, предметы инвентаря --- всё это может быть классами со своими свойствами и возможностями. Типы данных (такие как int, float и др.) в Python являются классами, структуры данных (dict, list, …) --- это также классы. print(int) print(dict)
Методы --- это функции, которые действуют в контексте экземпляра класса. Таким образом, они могут менять состояние экземпляра, обращаясь к атрибутам экземпляра или делать любую другую полезную работу.
Наследование – важная составляющая объектно-ориентированного программирования. Так или иначе мы уже сталкивались с ним, ведь объекты наследуют атрибуты своих классов. Однако обычно под наследованием в ООП понимается наличие классов и подклассов. Также их называют супер- или надклассами и классами, а также родительскими и дочерними классами.

Суть наследования здесь схожа с наследованием объектами от классов. Дочерние классы наследуют атрибуты родительских, а также могут переопределять атрибуты и добавлять свои.

Тема 5. Отладка и тестирование.
Скорее всего отладкой вы уже занимались, когда пытались выяснить, почему ваша программа не работает или работает некорректно. Если вы программируете в IDE, скорее всего, там есть инструментарий для отладки кода, и вы можете запускать вашу программу под отладчиком, ставить брейкпоинты, следить за переменными и т.д. Мы разберём классический механизм отладки с помощью Python Debugger-a. Например, мы написали программу, которая принимает на вход сайт, URL сайта и какую-то строчку, и ищет в коде сайта эту строчку и считает, сколько раз там встретилась эта строка. Затем запустили программу в Jupyter Notebook. Эта программа выдаёт неочевидную ошибку (проверьте, запустив самостоятельно), которую мы хотим отладить:

```
import re
import requests
def main(site_url, substring):
site_code = get_site_code(site_url)
matching_substrings = get_matching_substrings(site_code,
substring)
print('"{}" found {} times in {}'.format(
substring, len(matching_substrings), site_url
))
def get_site_code(site_url):
if not site_url.startswith('http'):
```

```
site_url = 'http://' + site_url
return requests.get(site_url).text
def get_matching_substrings(source, substring):
return re.findall(source, substring)
main('mail.ru', 'script')
```

Чтобы запустить отладчик в нашей программе, мы можем импортировать pdb и вызвать команду pdb.set_trace в том месте кода, где мы хотим остановить выполнение и начать отладку:

Настало время поговорить о тестировании, которого так бояться многие программисты. Если вы работаете над большим, быстро изменяющимся проектом с большим количеством разработчиков, вам нужно постоянно проверять, правильно ли работает ваша программа в различных условиях. Именно это и называется тестированием. Тестированию можно посвятить отдельную тему, курс и даже специализацию, потому что это огромная область. Мы с вами разберем наиболее популярный и распространенный вид тестирования — это unit-тестирование. Unit-тесты призваны протестировать небольшую функцию, класс или модуль — посмотреть, корректно ли он работает. Чтобы определить свой unittest можно воспользоваться стандартной библиотекой модулей unittest и определить свой класс, который наследуется от TestCase из модуля unittest. Внутри класса вы можете определить функции, которые и будут являться тестами. Каждая функция, которая начинается с test_ является тестом. В следующем примере мы хотим проверить, правильно ли у нас приводится типы, и например, корректно ли у нас работает функция get у пустого словаря. Делается это с помощью методов TestCase: assertEqual, assertIsNone, assertRaises и т.д. (подробнее можно прочесть в документации). Все они делают одно: проверяют, корректно ли работает выражение, правильно ли вызывается функция и так далее

Тема 6. Построение нейросети на Python.

Основные архитектуры нейронных сетей. Теорема Байеса. Алгоритм EM. Работа с математической библиотекой numpy. Введение в Tensor Flow и Keras.

Нейронные сети приобретают заслуженную популярность в последнее время всё больше и больше. Одним из простейших типов нейронных сетей является персептрон. В этой статье мы рассмотрим реализацию многослойного персептрона на Python с использованием надстройки для глубокого обучения Keras. Сперва скажем несколько слов о том, что такое многослойный персептрон и как он обучается. Многослойный персептрон Как известно, простейшим примером нейронной сети является многослойный (в частном случае, однослойный) персептрон. Рассмотрим, что собой представляет такая модель. Однослойный (многослойный) персептрон состоит из: – входного вектора (входные данные), – выходного вектора (выходные данные), – вектора(ов) промежуточного представления (скрытый(ые) слой). Элементы векторов принято еще называть нейронами. Вычисления в такой модели

распространяются от входа к выходу. Связям между нейронами на разных слоях соответствуют некоторые веса. Поэтому такая сеть является полносвязной.

Что происходит в одном нейроне В нейрон поступают входные значения, например, $x1$, $x2$, $x3$ (элементы входного вектора) с соответствующими весами $w1$, $w2$, $w3$. Далее, внутри нейрона происходит вычисление двух операций, а именно, композиции линейной и нелинейной функции: – сначала мы вычисляем взвешенную сумму входных значений и добавляем некоторый параметр смещения b, – далее, от полученных на предыдущем шаге значений, берём нелинейную функцию f, которую принято называть функцией активации. Таким образом, выходное значение h из одного нейрона вычислим по формуле $h = f\ (\sum wixi + b\ i\ )$. Вместе эти нейроны образуют большую нейронную сеть.

Обратим внимание на то, что одному слою в нейронной сети соответствует уже целая матрица весов $\boldsymbol{W}1$ и некоторый вектор смещений $\boldsymbol{B}$. Поэтому вычисления в одном слое персептрона можно представить в виде композиции матричного умножения, прибавления вектора смещений и поэлементного взятия нелинейной функции $\boldsymbol{H} = f(\boldsymbol{W1} \cdot \boldsymbol{X} + \boldsymbol{B1})$, где $\boldsymbol{H} = (\ h1\ h2\ h3\ h4\ )$, $\boldsymbol{W1} = (\ w11\ 1\ w12\ 1\ w13\ 1\ w21\ 1\ w22\ 1\ w23\ 1\ w31\ 1\ w32\ 1\ w33\ 1\ w41\ 1\ w42\ 1\ w43\ 1\ )$,$\boldsymbol{X} = (\ x1\ x2\ x3\ )$, $\boldsymbol{B1} = (\ b1\ b2\ b3\ b4\ )$, и $hi = f\ (\sum wij\ 1\ xj + bi\ j\ )$.

Процесс тренировки

Но как научить наш нейрон правильно отвечать на заданный вопрос? Для этого мы зададим каждому входящему сигналу вес, который может быть положительным или отрицательным числом. Если на входе будет сигнал с большим положительным весом или отрицательным весом, то это сильно повлияет на решение нейрона, которое он подаст на выход. Прежде чем мы начнем обучение модели, зададим для каждого примера случайное число в качестве веса. После этого мы можем приняться за тренировочный процесс, который будет выглядеть следующим образом:

В качестве входных данных мы возьмем примеры из тренировочного набора. Потом мы воспользуемся специальной формулой для расчета выхода нейрона, которая будет учитывать случайные веса, которые мы задали для каждого примера.

Далее посчитаем размер ошибки, который вычисляется как разница между числом, которое нейрон подал на выход и желаемым числом из примера.

В зависимости от того, в какую сторону нейрон ошибся, мы немного отрегулируем вес этого примера.

Повторим этот процесс 10 000 раз.

В какой-то момент веса достигнут оптимальных значений для тренировочного набора. Если после этого нейрону будет дана новая задача, которая следует такой же закономерности, он должен дать верный ответ.

Как написать это на Python

Хотя мы не будем использовать специальные библиотеки для нейронных сетей, мы импортируем следующие 4 метода из математической библиотеки numpy:

- exp — функция экспоненты
- array — метод создания матриц
- dot — метод перемножения матриц
- random — метод, подающий на выход случайное число

Теперь мы можем, например, представить наш тренировочный набор с использованием array():

```
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])=
training_set_outputs = array([[0, 1, 1, 0]]).T
```

Функция .T транспонирует матрицу из горизонтальной в вертикальную. В результате компьютер хранит эти числа таким образом:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Теперь мы готовы к более изящной версии кода. После нее добавим несколько финальных замечаний.

Обратите внимание, что на каждой итерации мы обрабатываем весь тренировочный набор одновременно. Таким образом наши переменные все являются матрицами.

Итак, вот полноценно работающий пример нейронной сети, написанный на Python:

```
from numpy import exp, array, random, dot

class NeuralNetwork():
def __init__(self):
Задаем порождающий элемент для генератора случайных чисел, чтобы он генерировал одинаковые числа при каждом запуске программы
random.seed(1)
```

Мы моделируем единственный нейрон с тремя входящими связями и одним выходом. Мы задаем случайные веса в матрице размера 3 x 1, где значения весов варьируются от -1 до 1, а среднее значение равно 0.
self.synaptic_weights = 2 * random.random((3, 1)) — 1

Функция сигмоиды, график которой имеет форму буквы S.
Мы используем эту функцию, чтобы нормализовать взвешенную сумму входных сигналов.
def __sigmoid(self, x):
return 1 / (1 + exp(-x))

Производная от функции сигмоиды. Это градиент ее кривой. Его значение указывает насколько нейронная сеть уверена в правильности существующего веса.
def __sigmoid_derivative(self, x):
return x * (1 — x)

Мы тренируем нейронную сеть методом проб и ошибок, каждый раз корректируя вес синапсов.
def train(self, training_set_inputs, training_set_outputs,
number_of_training_iterations):
for iteration in xrange(number_of_training_iterations):
Тренировочный набор передается нейронной сети (одному нейрону в нашем случае).
output = self.think(training_set_inputs)

Вычисляем ошибку (разницу между желаемым выходом и выходом, предсказанным нейроном).
error = training_set_outputs — output

Умножаем ошибку на входной сигнал и на градиент сигмоиды. В результате этого, те веса, в которых нейрон не уверен, будут откорректированы сильнее. Входные сигналы, которые равны нулю, не приводят к изменению веса.
adjustment = dot(training_set_inputs.T, error * self.__sigmoid_derivative(output))

Корректируем веса.
self.synaptic_weights += adjustment

Заставляем наш нейрон подумать.
def think(self, inputs):
Пропускаем входящие данные через нейрон.
return self.__sigmoid(dot(inputs, self.synaptic_weights))

if __name__ == «__main__»:

Инициализируем нейронную сеть, состоящую из одного нейрона.
neural_network = NeuralNetwork()

print «Random starting synaptic weights:
» print neural_network.synaptic_weights

Тренировочный набор для обучения. У нас это 4 примера, состоящих из 3 входящих значений и 1 выходящего значения.
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
training_set_outputs = array([[0, 1, 1, 0]]).T

Обучаем нейронную сеть на тренировочном наборе, повторяя процесс 10000 раз, каждый раз корректируя веса.
neural_network.train(training_set_inputs, training_set_outputs, 10000)

print «New synaptic weights after training:
» print neural_network.synaptic_weights

Тестируем нейрон на новом примере.
print «Considering new situation [1, 0, 0] -> ?:
» print neural_network.think(array([1, 0, 0]))

Этот код также можно найти на GitHub. Обратите внимание, что если вы используете Python 3, то вам будет нужно заменить команду “xrange” на “range”.