**顶峰科技123456项目代码**

import pygame
from pygame.sprite import Sprite
class Alien(Sprite):
"""单个外星人类"""
def \_\_init\_\_(self, ai\_settings, screen):
"""初始化外星人并设置起始位置"""
super(Alien, self).\_\_init\_\_()
self.ai\_settings = ai\_settings
self.screen = screen
# 加载外星人图像并设置其rect属性
self.image = pygame.image.load("images/alien.bmp")
self.rect = self.image.get\_rect()
# 每个外星人都在屏幕左上角附近
self.rect.x = self.rect.width
self.rect.y = self.rect.height
# 存储外星人位置
self.x = float(self.rect.x)
def blitme(self):
"""在指定位置绘制外星人"""
self.screen.blit(self.image, self.rect)
def update(self):
"""实现外星人移动"""
self.x += (self.ai\_settings.alien\_speed\_factor \*
self.ai\_settings.fleet\_direction)
self.rect.x = self.x
def check\_edges(self):
"""如果外星人位于屏幕边缘，就返回True"""
screen\_rect = self.screen.get\_rect()
if self.rect.right >= screen\_rect.right:
return True
elif self.rect.left <= 0:
return True
import pygame
import game\_functions as gf
from pygame.sprite import Group
from settings import Settings
from ship import Ship
from alien import Alien
from game\_stats import GameStats
from button import Button
from scoreboard import Scoreboard
def run\_game():
# 初始化游戏并创建一个屏幕对象
pygame.init()
ai\_settings = Settings()
screen = pygame.display.set\_mode(
(ai\_settings.screen\_width, ai\_settings.screen\_height))
pygame.display.set\_caption("Alien Invasion")
# 创建一艘飞船
ship = Ship(ai\_settings, screen)
# 创建一个用于存储子弹的编组
bullets = Group()
# 创建一个用于存储外星人的编组
aliens = Group()
# 创建外星人群
gf.create\_fleet(ai\_settings, screen, ship, aliens)
# 创建存储游戏统计信息的实例，并创建记分牌
stats = GameStats(ai\_settings)
sb = Scoreboard(ai\_settings, screen, stats)
# 创建play按钮
play\_button = Button(ai\_settings, screen, "Play")
# 开始游戏的主循环
while True:
# 监听鼠标和键盘时间
gf.check\_events(ai\_settings, screen, stats,
play\_button, sb, ship, aliens, bullets)
if stats.game\_active:
ship.update()
gf.update\_bullets(bullets, ai\_settings, screen,
stats, sb, ship, aliens)
gf.update\_aliens(ai\_settings, stats, screen, sb, ship, aliens, bullets)
# print(len(bullets))
# 更新屏幕上的图像，并切换到新屏幕
gf.update\_screen(ai\_settings, screen, stats, sb, ship,
aliens, bullets, play\_button)
run\_game()
import pygame
from pygame.sprite import Sprite
class Bullet(Sprite):
"""一个对飞船发射的子弹管理的类"""
def \_\_init\_\_(self,ai\_settings,screen,ship):
"""在飞船所在的位置创建一个子弹类"""
#print('子弹管理类开始构建')
super(Bullet,self).\_\_init\_\_()
self.screen = screen
#print('pygame.Rect 开始')
self.rect = pygame.Rect(0,0,ai\_settings.bullet\_width,ai\_settings.bullet\_height)
self.rect.centerx = ship.rect.centerx
self.rect.top = ship.rect.top
self.y = float(self.rect.y)
self.color = ai\_settings.bullet\_color
self.speed\_factor = ai\_settings.bullet\_speed\_factor
def update(self):
"""向上移动子弹"""
#更新表示子弹位置的小数值
self.y -= self.speed\_factor
#更新表示子弹的rect的位置
self.rect.y = self.y
def draw\_bullet(self):
"""在屏幕上绘制子弹"""
pygame.draw.rect(self.screen,self.color,self.rect)
import pygame.font
class Button():
"""按钮类"""
def \_\_init\_\_(self, ai\_settings, screen, msg):
self.ai\_settings = ai\_settings
self.screen = screen
self.screen\_rect = screen.get\_rect()
self.msg = msg
# 设置按钮的尺寸和其他属性
self.width, self.height = 200, 50
self.button\_color = (0,255,0)
self.text\_color = (255,255,255)
self.font = pygame.font.SysFont(None,48)
#创建按钮的rect对象，并使其居中
self.rect = pygame.Rect(0,0,self.width,self.height)
self.rect.center = self.screen\_rect.center
#按钮的标签
self.prep\_msg(msg)
def prep\_msg(self,msg):
"""将msg渲染为图像，并使其在按钮上居中"""
self.msg\_image = self.font.render(msg,True,self.text\_color,self.button\_color)
self.msg\_image\_rect = self.msg\_image.get\_rect()
self.msg\_image\_rect.center = self.rect.center
def draw\_button(self):
"""绘制一个使用颜色填充的按钮，在绘制文本"""
self.screen.fill(self.button\_color,self.rect)
self.screen.blit(self.msg\_image,self.msg\_image\_rect)
import sys
import time
import pygame
from bullet import Bullet
from alien import Alien
def check\_keydown\_events(event, ai\_settings, screen, ship, bullets):
"""按下键盘事件处理"""
if event.key == pygame.K\_RIGHT:
ship.rect.centerx += 1
ship.moving\_right = True
elif event.key == pygame.K\_LEFT:
ship.rect.centerx -= 1
ship.moving\_left = True
elif event.key == pygame.K\_SPACE:
# 发射子弹
fire\_bullet(ai\_settings, screen, ship, bullets)
elif event.key == pygame.K\_q:
sys.exit()
def check\_keyup\_events(event, ship):
"""键盘"""
if event.key == pygame.K\_RIGHT:
ship.moving\_right = False
if event.key == pygame.K\_LEFT:
ship.moving\_left = False
def check\_events(ai\_settings, screen, stats, play\_button, sb, ship, aliens, bullets):
"""监听鼠标和键盘事件"""
for event in pygame.event.get():
if event.type == pygame.QUIT:
sys.exit()
elif event.type == pygame.KEYDOWN:
check\_keydown\_events(event, ai\_settings, screen, ship, bullets)
elif event.type == pygame.KEYUP:
check\_keyup\_events(event, ship)
elif event.type == pygame.MOUSEBUTTONDOWN:
mouse\_x, mouse\_y = pygame.mouse.get\_pos()
check\_play\_button(ai\_settings, screen, stats, play\_button,
sb, ship, aliens, bullets, mouse\_x, mouse\_y)
def check\_play\_button(ai\_settings, screen, stats, play\_button, sb, ship, aliens, bullets, mouse\_x, mouse\_y):
"""在玩家点击play按钮时开始游戏"""
button\_checked = play\_button.rect.collidepoint(mouse\_x, mouse\_y)
if button\_checked and not stats.game\_active:
pygame.mouse.set\_visible(False)
ai\_settings.initialize\_dynamic\_settings()
# 重置游戏统计信息
stats.reset\_stats()
sb.prep\_score()
sb.prep\_level()
sb.prep\_ships()
stats.game\_active = True
# 清空外星人列表和子弹列表
aliens.empty()
bullets.empty()
# 创建一群新的外星人，并将飞船放到屏幕低端中央
create\_fleet(ai\_settings, screen, ship, aliens)
ship.center\_ship()
def update\_screen(ai\_settings, screen, stats, sb, ship, aliens, bullets, play\_button):
"""更新屏幕上的图像，并切换到新屏幕"""
# 每次循环时重绘屏幕
screen.fill(ai\_settings.bg\_color)
for bullet in bullets.sprites():
bullet.draw\_bullet()
ship.blitme()
aliens.draw(screen)
sb.show\_score()
if stats.game\_active == False:
play\_button.draw\_button()
# 让最近绘制的屏幕可见
pygame.display.flip()
def update\_bullets(bullets, ai\_settings, screen, stats, sb, ship, aliens):
bullets.update()
for bullet in bullets.copy():
if bullet.rect.bottom <= 0:
bullets.remove(bullet)
chect\_bullet\_alien\_collisions(
ai\_settings, screen, stats, sb, ship, aliens, bullets)
def chect\_bullet\_alien\_collisions(ai\_settings, screen, stats, sb, ship, aliens, bullets):
"""响应子弹和外星人的碰撞"""
# 检测是否有子弹击中了外星人
# 如果是这样，就删除相应的子弹和外星人
collisions = pygame.sprite.groupcollide(bullets, aliens, True, True)
if len(aliens) == 0:
bullets.empty()
ai\_settings.increase\_speed()
# 如果整群外星人被消灭，提高一个等级
stats.level += 1
sb.prep\_level()
create\_fleet(ai\_settings, screen, ship, aliens)
if collisions:
for aliens in collisions.values():
stats.score += ai\_settings.alien\_points \* len(aliens)
sb.prep\_score()
check\_high\_score(stats, sb)
def fire\_bullet(ai\_settings, screen, ship, bullets):
# 创建一个子弹并添加到bullets中
#print('new\_bullet start')
if len(bullets) < ai\_settings.bullet\_allowed:
new\_bullet = Bullet(ai\_settings, screen, ship)
#print("new\_bullet success")
bullets.add(new\_bullet)
def create\_fleet(ai\_settings, screen, ship, aliens):
"""创建外星人群"""
# 创建一个外星人，并计算一行可容纳多少个外星人
# 外星人间距为外星人宽度
alien = Alien(ai\_settings, screen)
number\_aliens\_x = get\_number\_alien\_x(ai\_settings, alien.rect.width)
number\_rows = get\_number\_rows(
ai\_settings, ship.rect.height, alien.rect.width)
for row\_number in range(number\_rows):
for alien\_number in range(number\_aliens\_x):
# 创建一个外星人并建其放置到当前行
create\_alien(ai\_settings, screen, aliens, alien\_number, row\_number)
def get\_number\_alien\_x(ai\_settings, alien\_width):
"""计算一行可容纳多少个外星人"""
available\_space\_x = ai\_settings.screen\_width - 2 \* alien\_width
number\_aliens\_x = int(available\_space\_x / (2 \* alien\_width))
return number\_aliens\_x
def create\_alien(ai\_settings, screen, aliens, alien\_number, row\_number):
"""创建一个外星人并建其放置到当前行"""
alien = Alien(ai\_settings, screen)
alien\_width = alien.rect.width
alien.x = alien\_width + 2 \* alien\_width \* alien\_number
alien.rect.x = alien.x
alien.rect.y = alien.rect.height + 2 \* alien.rect.height \* row\_number
aliens.add(alien)
def get\_number\_rows(ai\_settings, ship\_height, alien\_height):
"""计算屏幕可容纳多上行外星人"""
avialable\_space\_y = (ai\_settings.screen\_height -
(3 \* alien\_height) - ship\_height)
number\_rows = int(avialable\_space\_y / (2 \* alien\_height))
return number\_rows
def update\_aliens(ai\_settings, stats, screen, sb, ship, aliens, bullets):
"""更新外星人群中所有外星人的位置"""
check\_fleet\_edges(ai\_settings, aliens)
aliens.update()
# 检测外星人和飞船的碰撞
if pygame.sprite.spritecollideany(ship, aliens):
ship\_hit(ai\_settings, stats, screen, sb, ship, aliens, bullets)
check\_alien\_bottom(ai\_settings, stats, screen, sb, ship, aliens, bullets)
def check\_fleet\_edges(ai\_settings, aliens):
"""有外星人到达边缘时采取相应的措施"""
for alien in aliens.sprites():
if alien.check\_edges():
change\_fleet\_direction(ai\_settings, aliens)
break
def change\_fleet\_direction(ai\_settings, aliens):
"""将外星人群向下移，并改变它们的方向"""
for alien in aliens.sprites():
alien.rect.y += ai\_settings.alien\_drop\_speed
ai\_settings.fleet\_direction \*= -1
def ship\_hit(ai\_settings, stats, screen, sb, ship, aliens, bullets):
"""响应被外星人撞到飞船"""
print("ships\_left->"+str(stats.ships\_left))
if stats.ships\_left > 1:
# 将ships\_left 减1
stats.ships\_left -= 1
# 更新飞船记录
sb.prep\_ships()
# 清空外星人列表和子弹列表
aliens.empty()
bullets.empty()
# 创建一群新的外星人，并将飞船放到屏幕低端中央
create\_fleet(ai\_settings, screen, ship, aliens)
ship.center\_ship()
# 暂停
time.sleep(0.5)
else:
stats.game\_active = False
pygame.mouse.set\_visible(True)
def check\_alien\_bottom(ai\_settings, stats, screen, sb, ship, aliens, bullets):
"""检测是否有外星人到达了屏幕低端"""
screen\_rect = screen.get\_rect()
for alien in aliens.sprites():
if alien.rect.bottom >= screen\_rect.bottom:
ship\_hit(ai\_settings, stats, screen, sb, ship, aliens, bullets)
break
def check\_high\_score(stats, sb):
"""检测是否诞生了新的最高分"""
if stats.score > stats.high\_score:
stats.high\_score = stats.score
sb.prep\_high\_score()
import pygame
from settings import Settings
class GameStats():
"""跟踪游戏的统计信息"""
def \_\_init\_\_(self,ai\_settings):
self.ai\_settings = ai\_settings
self.reset\_stats()
self.game\_active = False
self.high\_score = 0
def reset\_stats(self):
"""初始化在游戏运行期间可能变化的统计信息"""
self.ships\_left = self.ai\_settings.ship\_limit
self.score = 0
self.level = 1
import pygame.font
from pygame.sprite import Group
from ship import Ship
class Scoreboard():
"""显示得分信息的类"""
def \_\_init\_\_(self, ai\_settings, screen, stats):
self.screen = screen
self.screen\_rect = screen.get\_rect()
self.ai\_settings = ai\_settings
self.stats = stats
# 显示得分信息时使用的字体信息
self.text\_color = (30, 30, 30)
self.font = pygame.font.SysFont(None, 48)
# 准备初始得分图像
self.prep\_score()
# 最高得分，在任何情况下都不能重置最高得分
self.prep\_high\_score()
# 记录等级
self.prep\_level()
self.prep\_ships()
def prep\_score(self):
"""将得分转换为一副渲染的图像"""
rounded\_score = int(round(self.stats.score, -1))
score\_str = "{:,}".format(rounded\_score)
self.score\_image = self.font.render(
score\_str, True, self.text\_color, self.ai\_settings.bg\_color)
# 将得分放到屏幕右上角
self.score\_rect = self.score\_image.get\_rect()
self.score\_rect.right = self.screen\_rect.right
self.score\_rect.top = 30
def prep\_high\_score(self):
"""将最高得分转换为一副渲染的图像"""
high\_score = int(round(self.stats.score, -1))
high\_score\_str = "{:,}".format(high\_score)
self.high\_score\_image = self.font.render(
high\_score\_str, True, self.text\_color, self.ai\_settings.bg\_color)
# 将得分放到屏幕右上角
self.high\_score\_rect = self.high\_score\_image .get\_rect()
self.high\_score\_rect.centerx = self.screen\_rect.centerx
self.high\_score\_rect.top = self.screen\_rect.top
def prep\_level(self):
"""将等级转换为一副渲染的图像"""
self.level\_image = self.font.render(
str(self.stats.level), True, self.text\_color, self.ai\_settings.bg\_color)
# 将得分放到屏幕右上角
self.level\_rect = self.score\_image.get\_rect()
self.level\_rect.right = self.score\_rect.right
self.level\_rect.top = self.score\_rect.bottom + 10
def prep\_ships(self):
"""显示还余下多少艘飞船"""
self.ships = Group()
for ship\_number in range(self.stats.ships\_left):
ship = Ship(self.ai\_settings, self.screen)
ship.rect.x = 10 + ship\_number \* ship.rect.width
ship.rect.y = 10
self.ships.add(ship)
def show\_score(self):
"""在屏幕上显示得分"""
self.screen.blit(self.score\_image, self.score\_rect)
self.screen.blit(self.high\_score\_image, self.high\_score\_rect)
self.screen.blit(self.level\_image, self.level\_rect)
self.ships.draw(self.screen)
class Settings():
def \_\_init\_\_(self):
"""初始化游戏的设置"""
# 屏幕设置
self.screen\_width = 800
self.screen\_height = 600
self.bg\_color = (230, 230, 230)
# 飞船设置
self.ship\_speed\_factor = 1.5
self.ship\_limit = 3
# 子弹设置
self.bullet\_speed\_factor = 3
self.bullet\_width = 3
self.bullet\_height = 15
self.bullet\_color = 60, 60, 60
self.bullet\_allowed = 20
# 外星人设置
self.alien\_speed\_factor = 1
self.alien\_drop\_speed = 10
# fleet\_direction为1表示向右移动，为-1表示向左移动
self.fleet\_direction = 1
# 以什么样的速度加快游戏节奏
self.speedup\_scale = 1.1
#外星人点数提高速度
self.score\_scale = 1.5
self.alien\_points = 50
self.initialize\_dynamic\_settings()
def initialize\_dynamic\_settings(self):
self.ship\_speed\_factor = 1.5
self.bullet\_speed\_factor = 3
self.alien\_speed\_factor = 1
self.fleet\_direction = 1
self.alien\_points = int(self.alien\_points \* self.score\_scale)
def increase\_speed(self):
self.ship\_speed\_factor \*= self.speedup\_scale
self.bullet\_speed\_factor = self.speedup\_scale
self.alien\_speed\_factor = self.speedup\_scale
self.alien\_points = int(self.alien\_points \* self.score\_scale)
#print(self.alien\_points)
import pygame
from pygame.sprite import Sprite
class Ship(Sprite):
"""初始化飞船并设置其初始位置"""
def \_\_init\_\_(self,ai\_settings,screen):
"""初始化飞船，并设置其初始位置"""
super(Ship,self).\_\_init\_\_()
self.screen = screen
self.ai\_settings = ai\_settings
#加载飞船图像并获取其外接矩形
self.image = pygame.image.load("images/ship.bmp")
self.rect = self.image.get\_rect()
self.screen\_rect = screen.get\_rect()
#将每部飞船放到屏幕底部中央
self.rect.centerx = self.screen\_rect.centerx
self.rect.bottom = self.screen\_rect.bottom
self.center = float(self.screen\_rect.centerx)
self.moving\_right = False
self.moving\_left = False
def update(self):
"""根据移动标志调整飞船位置"""
if self.moving\_right and self.rect.right < self.screen\_rect.right:
self.center += self.ai\_settings.ship\_speed\_factor
if self.moving\_left and self.rect.left > 0:
self.center -= self.ai\_settings.ship\_speed\_factor
self.rect.centerx = self.center
def blitme(self):
"""在指定位置绘制飞船"""
self.screen.blit(self.image,self.rect)
def center\_ship(self):
"""让飞船在屏幕上居中"""
self.center = self.screen\_rect.centerx
#!/usr/bin/env python
import os
import sys
if \_\_name\_\_ == '\_\_main\_\_':
os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'learning\_log.settings')
try:
from django.core.management import execute\_from\_command\_line
except ImportError as exc:
raise ImportError(
"Couldn't import Django. Are you sure it's installed and "
"available on your PYTHONPATH environment variable? Did you "
"forget to activate a virtual environment?"
) from exc
execute\_from\_command\_line(sys.argv)
"""
Django settings for learning\_log project.
Generated by 'django-admin startproject' using Django 2.1.5.
For more information on this file, see
https://docs.djangoproject.com/en/2.1/topics/settings/
For the full list of settings and their values, see
https://docs.djangoproject.com/en/2.1/ref/settings/
"""
import os
# Build paths inside the project like this: os.path.join(BASE\_DIR, ...)
BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))
# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/
# SECURITY WARNING: keep the secret key used in production secret!
SECRET\_KEY = 'v@6-#l7-fxpzeez6!h8ju3()t0bl7z!b-\_2-8s#2se4o#l+@+w'
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True
ALLOWED\_HOSTS = []
# Application definition
INSTALLED\_APPS = [
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'learning\_logs'
]
MIDDLEWARE = [
'django.middleware.security.SecurityMiddleware',
'django.contrib.sessions.middleware.SessionMiddleware',
'django.middleware.common.CommonMiddleware',
'django.middleware.csrf.CsrfViewMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.contrib.messages.middleware.MessageMiddleware',
'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
ROOT\_URLCONF = 'learning\_log.urls'
TEMPLATES = [
{
'BACKEND': 'django.template.backends.django.DjangoTemplates',
'DIRS': [],
'APP\_DIRS': True,
'OPTIONS': {
'context\_processors': [
'django.template.context\_processors.debug',
'django.template.context\_processors.request',
'django.contrib.auth.context\_processors.auth',
'django.contrib.messages.context\_processors.messages',
],
},
},
]
WSGI\_APPLICATION = 'learning\_log.wsgi.application'
# Database
# https://docs.djangoproject.com/en/2.1/ref/settings/#databases
DATABASES = {
'default': {
'ENGINE': 'django.db.backends.sqlite3',
'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),
}
}
# Password validation
# https://docs.djangoproject.com/en/2.1/ref/settings/#auth-password-validators
AUTH\_PASSWORD\_VALIDATORS = [
{
'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',
},
{
'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',
},
{
'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',
},
{
'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',
},
]
# Internationalization
# https://docs.djangoproject.com/en/2.1/topics/i18n/
LANGUAGE\_CODE = 'en-us'
TIME\_ZONE = 'UTC'
USE\_I18N = True
USE\_L10N = True
USE\_TZ = True
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/2.1/howto/static-files/
STATIC\_URL = '/static/'
"""learning\_log URL Configuration
The `urlpatterns` list routes URLs to views. For more information please see:
https://docs.djangoproject.com/en/2.1/topics/http/urls/
Examples:
Function views
1. Add an import: from my\_app import views
2. Add a URL to urlpatterns: path('', views.home, name='home')
Class-based views
1. Add an import: from other\_app.views import Home
2. Add a URL to urlpatterns: path('', Home.as\_view(), name='home')
Including another URLconf
1. Import the include() function: from django.urls import include, path
2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path
urlpatterns = [
path('admin/', admin.site.urls),
]
"""
WSGI config for learning\_log project.
It exposes the WSGI callable as a module-level variable named ``application``.
For more information on this file, see
https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi/
"""
import os
from django.core.wsgi import get\_wsgi\_application
os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'learning\_log.settings')
application = get\_wsgi\_application()
from django.contrib import admin
# Register your models here.
from learning\_logs.models import Topic
from learning\_logs.models import Entry
admin.site.register(Topic)
admin.site.register(Entry)
from django.apps import AppConfig
class LearningLogsConfig(AppConfig):
name = 'learning\_logs'
from django.db import models
# Create your models here.
class Topic(models.Model):
"""用户学习相关主题"""
text = models.CharField(max\_length = 200)
data\_added = models.DateTimeField(auto\_now\_add=True)
def \_\_str\_\_(self):
"""返回模型的字符串表示"""
return self.text
class Entry(models.Model):
"""学到的有关主题的所有知识"""
topic = models.ForeignKey('Topic',on\_delete=models.CASCADE)
text = models.TextField()
data\_added = models.DateTimeField(auto\_now\_add=True)
class Mate:
verbose\_name\_plural = 'entries'
def \_\_str\_\_(self):
"""返回模型的字符串表示"""
return self.text[:50]+ "..."
from django.test import TestCase
# Create your tests here.
from django.shortcuts import render
# Create your views here.
#!D:\python\learning\_log\11\_env\Scripts\python.exe
# EASY-INSTALL-ENTRY-SCRIPT: 'autopep8==1.4.3','console\_scripts','autopep8'
\_\_requires\_\_ = 'autopep8==1.4.3'
import re
import sys
from pkg\_resources import load\_entry\_point
if \_\_name\_\_ == '\_\_main\_\_':
sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$', '', sys.argv[0])
sys.exit(
load\_entry\_point('autopep8==1.4.3', 'console\_scripts', 'autopep8')()
)
#!D:\python\learning\_log\11\_env\Scripts\python.exe
from django.core import management
if \_\_name\_\_ == "\_\_main\_\_":
management.execute\_from\_command\_line()
# Generated by Django 2.1.5 on 2019-02-07 14:30
from django.db import migrations, models
class Migration(migrations.Migration):
initial = True
dependencies = [
]
operations = [
migrations.CreateModel(
name='Topic',
fields=[
('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),
('text', models.CharField(max\_length=200)),
('data\_added', models.DateTimeField(auto\_now\_add=True)),
],
),
]
# Generated by Django 2.1.5 on 2019-02-08 02:52
from django.db import migrations, models
import django.db.models.deletion
class Migration(migrations.Migration):
dependencies = [
('learning\_logs', '0001\_initial'),
]
operations = [
migrations.CreateModel(
name='Entry',
fields=[
('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),
('text', models.TextField()),
('data\_added', models.DateTimeField(auto\_now\_add=True)),
('topic', models.ForeignKey(on\_delete=django.db.models.deletion.CASCADE, to='learning\_logs.Topic')),
],
),
]
#!/usr/bin/env python
# Copyright (C) 2010-2011 Hideo Hattori
# Copyright (C) 2011-2013 Hideo Hattori, Steven Myint
# Copyright (C) 2013-2016 Hideo Hattori, Steven Myint, Bill Wendling
#
# Permission is hereby granted, free of charge, to any person obtaining
# a copy of this software and associated documentation files (the
# "Software"), to deal in the Software without restriction, including
# without limitation the rights to use, copy, modify, merge, publish,
# distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to
# the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
# BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
# ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
"""Automatically formats Python code to conform to the PEP 8 style guide.
Fixes that only need be done once can be added by adding a function of the form
"fix\_<code>(source)" to this module. They should return the fixed source code.
These fixes are picked up by apply\_global\_fixes().
Fixes that depend on pycodestyle should be added as methods to FixPEP8. See the
class documentation for more information.
"""
from \_\_future\_\_ import absolute\_import
from \_\_future\_\_ import division
from \_\_future\_\_ import print\_function
from \_\_future\_\_ import unicode\_literals
import argparse
import codecs
import collections
import copy
import difflib
import fnmatch
import inspect
import io
import keyword
import locale
import os
import re
import signal
import sys
import textwrap
import token
import tokenize
import pycodestyle
try:
unicode
except NameError:
unicode = str
\_\_version\_\_ = '1.4.3'
CR = '\r'
LF = '\n'
CRLF = '\r\n'
PYTHON\_SHEBANG\_REGEX = re.compile(r'^#!.\*\bpython[23]?\b\s\*$')
LAMBDA\_REGEX = re.compile(r'([\w.]+)\s=\slambda\s\*([\(\)=\w,\s.]\*):')
COMPARE\_NEGATIVE\_REGEX = re.compile(r'\b(not)\s+([^][)(}{]+?)\s+(in|is)\s')
COMPARE\_NEGATIVE\_REGEX\_THROUGH = re.compile(r'\b(not\s+in|is\s+not)\s')
BARE\_EXCEPT\_REGEX = re.compile(r'except\s\*:')
STARTSWITH\_DEF\_REGEX = re.compile(r'^(async\s+def|def)\s.\*\):')
EXIT\_CODE\_OK = 0
EXIT\_CODE\_ERROR = 1
EXIT\_CODE\_EXISTS\_DIFF = 2
# For generating line shortening candidates.
SHORTEN\_OPERATOR\_GROUPS = frozenset([
frozenset([',']),
frozenset(['%']),
frozenset([',', '(', '[', '{']),
frozenset(['%', '(', '[', '{']),
frozenset([',', '(', '[', '{', '%', '+', '-', '\*', '/', '//']),
frozenset(['%', '+', '-', '\*', '/', '//']),
])
DEFAULT\_IGNORE = 'E226,E24,W50,W690' # TODO: use pycodestyle.DEFAULT\_IGNORE
DEFAULT\_INDENT\_SIZE = 4
SELECTED\_GLOBAL\_FIXED\_METHOD\_CODES = ['W602', ]
# W602 is handled separately due to the need to avoid "with\_traceback".
CODE\_TO\_2TO3 = {
'E231': ['ws\_comma'],
'E721': ['idioms'],
'W601': ['has\_key'],
'W603': ['ne'],
'W604': ['repr'],
'W690': ['apply',
'except',
'exitfunc',
'numliterals',
'operator',
'paren',
'reduce',
'renames',
'standarderror',
'sys\_exc',
'throw',
'tuple\_params',
'xreadlines']}
if sys.platform == 'win32': # pragma: no cover
DEFAULT\_CONFIG = os.path.expanduser(r'~\.pycodestyle')
else:
DEFAULT\_CONFIG = os.path.join(os.getenv('XDG\_CONFIG\_HOME') or
os.path.expanduser('~/.config'),
'pycodestyle')
# fallback, use .pep8
if not os.path.exists(DEFAULT\_CONFIG): # pragma: no cover
if sys.platform == 'win32':
DEFAULT\_CONFIG = os.path.expanduser(r'~\.pep8')
else:
DEFAULT\_CONFIG = os.path.join(os.path.expanduser('~/.config'), 'pep8')
PROJECT\_CONFIG = ('setup.cfg', 'tox.ini', '.pep8', '.flake8')
MAX\_PYTHON\_FILE\_DETECTION\_BYTES = 1024
def open\_with\_encoding(filename,
encoding=None, mode='r', limit\_byte\_check=-1):
"""Return opened file with a specific encoding."""
if not encoding:
encoding = detect\_encoding(filename, limit\_byte\_check=limit\_byte\_check)
return io.open(filename, mode=mode, encoding=encoding,
newline='') # Preserve line endings
def detect\_encoding(filename, limit\_byte\_check=-1):
"""Return file encoding."""
try:
with open(filename, 'rb') as input\_file:
from lib2to3.pgen2 import tokenize as lib2to3\_tokenize
encoding = lib2to3\_tokenize.detect\_encoding(input\_file.readline)[0]
with open\_with\_encoding(filename, encoding) as test\_file:
test\_file.read(limit\_byte\_check)
return encoding
except (LookupError, SyntaxError, UnicodeDecodeError):
return 'latin-1'
def readlines\_from\_file(filename):
"""Return contents of file."""
with open\_with\_encoding(filename) as input\_file:
return input\_file.readlines()
def extended\_blank\_lines(logical\_line,
blank\_lines,
blank\_before,
indent\_level,
previous\_logical):
"""Check for missing blank lines after class declaration."""
if previous\_logical.startswith('def '):
if blank\_lines and pycodestyle.DOCSTRING\_REGEX.match(logical\_line):
yield (0, 'E303 too many blank lines ({})'.format(blank\_lines))
elif pycodestyle.DOCSTRING\_REGEX.match(previous\_logical):
# Missing blank line between class docstring and method declaration.
if (
indent\_level and
not blank\_lines and
not blank\_before and
logical\_line.startswith(('def ')) and
'(self' in logical\_line
):
yield (0, 'E301 expected 1 blank line, found 0')
pycodestyle.register\_check(extended\_blank\_lines)
def continued\_indentation(logical\_line, tokens, indent\_level, hang\_closing,
indent\_char, noqa):
"""Override pycodestyle's function to provide indentation information."""
first\_row = tokens[0][2][0]
nrows = 1 + tokens[-1][2][0] - first\_row
if noqa or nrows == 1:
return
# indent\_next tells us whether the next block is indented. Assuming
# that it is indented by 4 spaces, then we should not allow 4-space
# indents on the final continuation line. In turn, some other
# indents are allowed to have an extra 4 spaces.
indent\_next = logical\_line.endswith(':')
row = depth = 0
valid\_hangs = (
(DEFAULT\_INDENT\_SIZE,)
if indent\_char != '\t' else (DEFAULT\_INDENT\_SIZE,
2 \* DEFAULT\_INDENT\_SIZE)
)
# Remember how many brackets were opened on each line.
parens = [0] \* nrows
# Relative indents of physical lines.
rel\_indent = [0] \* nrows
# For each depth, collect a list of opening rows.
open\_rows = [[0]]
# For each depth, memorize the hanging indentation.
hangs = [None]
# Visual indents.
indent\_chances = {}
last\_indent = tokens[0][2]
indent = [last\_indent[1]]
last\_token\_multiline = None
line = None
last\_line = ''
last\_line\_begins\_with\_multiline = False
for token\_type, text, start, end, line in tokens:
newline = row < start[0] - first\_row
if newline:
row = start[0] - first\_row
newline = (not last\_token\_multiline and
token\_type not in (tokenize.NL, tokenize.NEWLINE))
last\_line\_begins\_with\_multiline = last\_token\_multiline
if newline:
# This is the beginning of a continuation line.
last\_indent = start
# Record the initial indent.
rel\_indent[row] = pycodestyle.expand\_indent(line) - indent\_level
# Identify closing bracket.
close\_bracket = (token\_type == tokenize.OP and text in ']})')
# Is the indent relative to an opening bracket line?
for open\_row in reversed(open\_rows[depth]):
hang = rel\_indent[row] - rel\_indent[open\_row]
hanging\_indent = hang in valid\_hangs
if hanging\_indent:
break
if hangs[depth]:
hanging\_indent = (hang == hangs[depth])
visual\_indent = (not close\_bracket and hang > 0 and
indent\_chances.get(start[1]))
if close\_bracket and indent[depth]:
# Closing bracket for visual indent.
if start[1] != indent[depth]:
yield (start, 'E124 {}'.format(indent[depth]))
elif close\_bracket and not hang:
# closing bracket matches indentation of opening bracket's line
if hang\_closing:
yield (start, 'E133 {}'.format(indent[depth]))
elif indent[depth] and start[1] < indent[depth]:
# Visual indent is broken.
yield (start, 'E128 {}'.format(indent[depth]))
elif (hanging\_indent or
(indent\_next and
rel\_indent[row] == 2 \* DEFAULT\_INDENT\_SIZE)):
# Hanging indent is verified.
if close\_bracket and not hang\_closing:
yield (start, 'E123 {}'.format(indent\_level +
rel\_indent[open\_row]))
hangs[depth] = hang
elif visual\_indent is True:
# Visual indent is verified.
indent[depth] = start[1]
elif visual\_indent in (text, unicode):
# Ignore token lined up with matching one from a previous line.
pass
else:
one\_indented = (indent\_level + rel\_indent[open\_row] +
DEFAULT\_INDENT\_SIZE)
# Indent is broken.
if hang <= 0:
error = ('E122', one\_indented)
elif indent[depth]:
error = ('E127', indent[depth])
elif not close\_bracket and hangs[depth]:
error = ('E131', one\_indented)
elif hang > DEFAULT\_INDENT\_SIZE:
error = ('E126', one\_indented)
else:
hangs[depth] = hang
error = ('E121', one\_indented)
yield (start, '{} {}'.format(\*error))
# Look for visual indenting.
if (
parens[row] and
token\_type not in (tokenize.NL, tokenize.COMMENT) and
not indent[depth]
):
indent[depth] = start[1]
indent\_chances[start[1]] = True
# Deal with implicit string concatenation.
elif (token\_type in (tokenize.STRING, tokenize.COMMENT) or
text in ('u', 'ur', 'b', 'br')):
indent\_chances[start[1]] = unicode
# Special case for the "if" statement because len("if (") is equal to
# 4.
elif not indent\_chances and not row and not depth and text == 'if':
indent\_chances[end[1] + 1] = True
elif text == ':' and line[end[1]:].isspace():
open\_rows[depth].append(row)
# Keep track of bracket depth.
if token\_type == tokenize.OP:
if text in '([{':
depth += 1
indent.append(0)
hangs.append(None)
if len(open\_rows) == depth:
open\_rows.append([])
open\_rows[depth].append(row)
parens[row] += 1
elif text in ')]}' and depth > 0:
# Parent indents should not be more than this one.
prev\_indent = indent.pop() or last\_indent[1]
hangs.pop()
for d in range(depth):
if indent[d] > prev\_indent:
indent[d] = 0
for ind in list(indent\_chances):
if ind >= prev\_indent:
del indent\_chances[ind]
del open\_rows[depth + 1:]
depth -= 1
if depth:
indent\_chances[indent[depth]] = True
for idx in range(row, -1, -1):
if parens[idx]:
parens[idx] -= 1
break
assert len(indent) == depth + 1
if (
start[1] not in indent\_chances and
# This is for purposes of speeding up E121 (GitHub #90).
not last\_line.rstrip().endswith(',')
):
# Allow to line up tokens.
indent\_chances[start[1]] = text
last\_token\_multiline = (start[0] != end[0])
if last\_token\_multiline:
rel\_indent[end[0] - first\_row] = rel\_indent[row]
last\_line = line
if (
indent\_next and
not last\_line\_begins\_with\_multiline and
pycodestyle.expand\_indent(line) == indent\_level + DEFAULT\_INDENT\_SIZE
):
pos = (start[0], indent[0] + 4)
desired\_indent = indent\_level + 2 \* DEFAULT\_INDENT\_SIZE
if visual\_indent:
yield (pos, 'E129 {}'.format(desired\_indent))
else:
yield (pos, 'E125 {}'.format(desired\_indent))
del pycodestyle.\_checks['logical\_line'][pycodestyle.continued\_indentation]
pycodestyle.register\_check(continued\_indentation)
class FixPEP8(object):
"""Fix invalid code.
Fixer methods are prefixed "fix\_". The \_fix\_source() method looks for these
automatically.
The fixer method can take either one or two arguments (in addition to
self). The first argument is "result", which is the error information from
pycodestyle. The second argument, "logical", is required only for
logical-line fixes.
The fixer method can return the list of modified lines or None. An empty
list would mean that no changes were made. None would mean that only the
line reported in the pycodestyle error was modified. Note that the modified
line numbers that are returned are indexed at 1. This typically would
correspond with the line number reported in the pycodestyle error
information.
[fixed method list]
- e111,e114,e115,e116
- e121,e122,e123,e124,e125,e126,e127,e128,e129
- e201,e202,e203
- e211
- e221,e222,e223,e224,e225
- e231
- e251,e252
- e261,e262
- e271,e272,e273,e274
- e301,e302,e303,e304,e305,e306
- e401,e402
- e502
- e701,e702,e703,e704
- e711,e712,e713,e714
- e722
- e731
- w291
- w503,504
"""
def \_\_init\_\_(self, filename,
options,
contents=None,
long\_line\_ignore\_cache=None):
self.filename = filename
if contents is None:
self.source = readlines\_from\_file(filename)
else:
sio = io.StringIO(contents)
self.source = sio.readlines()
self.options = options
self.indent\_word = \_get\_indentword(''.join(self.source))
# collect imports line
self.imports = {}
for i, line in enumerate(self.source):
if (line.find("import ") == 0 or line.find("from ") == 0) and \
line not in self.imports:
# collect only import statements that first appeared
self.imports[line] = i
self.long\_line\_ignore\_cache = (
set() if long\_line\_ignore\_cache is None
else long\_line\_ignore\_cache)
# Many fixers are the same even though pycodestyle categorizes them
# differently.
self.fix\_e115 = self.fix\_e112
self.fix\_e121 = self.\_fix\_reindent
self.fix\_e122 = self.\_fix\_reindent
self.fix\_e123 = self.\_fix\_reindent
self.fix\_e124 = self.\_fix\_reindent
self.fix\_e126 = self.\_fix\_reindent
self.fix\_e127 = self.\_fix\_reindent
self.fix\_e128 = self.\_fix\_reindent
self.fix\_e129 = self.\_fix\_reindent
self.fix\_e133 = self.fix\_e131
self.fix\_e202 = self.fix\_e201
self.fix\_e203 = self.fix\_e201
self.fix\_e211 = self.fix\_e201
self.fix\_e221 = self.fix\_e271
self.fix\_e222 = self.fix\_e271
self.fix\_e223 = self.fix\_e271
self.fix\_e226 = self.fix\_e225
self.fix\_e227 = self.fix\_e225
self.fix\_e228 = self.fix\_e225
self.fix\_e241 = self.fix\_e271
self.fix\_e242 = self.fix\_e224
self.fix\_e252 = self.fix\_e225
self.fix\_e261 = self.fix\_e262
self.fix\_e272 = self.fix\_e271
self.fix\_e273 = self.fix\_e271
self.fix\_e274 = self.fix\_e271
self.fix\_e306 = self.fix\_e301
self.fix\_e501 = (
self.fix\_long\_line\_logically if
options and (options.aggressive >= 2 or options.experimental) else
self.fix\_long\_line\_physically)
self.fix\_e703 = self.fix\_e702
self.fix\_w293 = self.fix\_w291
def \_fix\_source(self, results):
try:
(logical\_start, logical\_end) = \_find\_logical(self.source)
logical\_support = True
except (SyntaxError, tokenize.TokenError): # pragma: no cover
logical\_support = False
completed\_lines = set()
for result in sorted(results, key=\_priority\_key):
if result['line'] in completed\_lines:
continue
fixed\_methodname = 'fix\_' + result['id'].lower()
if hasattr(self, fixed\_methodname):
fix = getattr(self, fixed\_methodname)
line\_index = result['line'] - 1
original\_line = self.source[line\_index]
is\_logical\_fix = len(\_get\_parameters(fix)) > 2
if is\_logical\_fix:
logical = None
if logical\_support:
logical = \_get\_logical(self.source,
result,
logical\_start,
logical\_end)
if logical and set(range(
logical[0][0] + 1,
logical[1][0] + 1)).intersection(
completed\_lines):
continue
modified\_lines = fix(result, logical)
else:
modified\_lines = fix(result)
if modified\_lines is None:
# Force logical fixes to report what they modified.
assert not is\_logical\_fix
if self.source[line\_index] == original\_line:
modified\_lines = []
if modified\_lines:
completed\_lines.update(modified\_lines)
elif modified\_lines == []: # Empty list means no fix
if self.options.verbose >= 2:
print(
'---> Not fixing {error} on line {line}'.format(
error=result['id'], line=result['line']),
file=sys.stderr)
else: # We assume one-line fix when None.
completed\_lines.add(result['line'])
else:
if self.options.verbose >= 3:
print(
"---> '{}' is not defined.".format(fixed\_methodname),
file=sys.stderr)
info = result['info'].strip()
print('---> {}:{}:{}:{}'.format(self.filename,
result['line'],
result['column'],
info),
file=sys.stderr)
def fix(self):
"""Return a version of the source code with PEP 8 violations fixed."""
pep8\_options = {
'ignore': self.options.ignore,
'select': self.options.select,
'max\_line\_length': self.options.max\_line\_length,
'hang\_closing': self.options.hang\_closing,
}
results = \_execute\_pep8(pep8\_options, self.source)
if self.options.verbose:
progress = {}
for r in results:
if r['id'] not in progress:
progress[r['id']] = set()
progress[r['id']].add(r['line'])
print('---> {n} issue(s) to fix {progress}'.format(
n=len(results), progress=progress), file=sys.stderr)
if self.options.line\_range:
start, end = self.options.line\_range
results = [r for r in results
if start <= r['line'] <= end]
self.\_fix\_source(filter\_results(source=''.join(self.source),
results=results,
aggressive=self.options.aggressive))
if self.options.line\_range:
# If number of lines has changed then change line\_range.
count = sum(sline.count('\n')
for sline in self.source[start - 1:end])
self.options.line\_range[1] = start + count - 1
return ''.join(self.source)
def \_fix\_reindent(self, result):
"""Fix a badly indented line.
This is done by adding or removing from its initial indent only.
"""
num\_indent\_spaces = int(result['info'].split()[1])
line\_index = result['line'] - 1
target = self.source[line\_index]
self.source[line\_index] = ' ' \* num\_indent\_spaces + target.lstrip()
def fix\_e112(self, result):
"""Fix under-indented comments."""
line\_index = result['line'] - 1
target = self.source[line\_index]
if not target.lstrip().startswith('#'):
# Don't screw with invalid syntax.
return []
self.source[line\_index] = self.indent\_word + target
def fix\_e113(self, result):
"""Fix unexpected indentation."""
line\_index = result['line'] - 1
target = self.source[line\_index]
indent = \_get\_indentation(target)
stripped = target.lstrip()
self.source[line\_index] = indent[1:] + stripped
def fix\_e116(self, result):
"""Fix over-indented comments."""
line\_index = result['line'] - 1
target = self.source[line\_index]
indent = \_get\_indentation(target)
stripped = target.lstrip()
if not stripped.startswith('#'):
# Don't screw with invalid syntax.
return []
self.source[line\_index] = indent[1:] + stripped
def fix\_e125(self, result):
"""Fix indentation undistinguish from the next logical line."""
num\_indent\_spaces = int(result['info'].split()[1])
line\_index = result['line'] - 1
target = self.source[line\_index]
spaces\_to\_add = num\_indent\_spaces - len(\_get\_indentation(target))
indent = len(\_get\_indentation(target))
modified\_lines = []
while len(\_get\_indentation(self.source[line\_index])) >= indent:
self.source[line\_index] = (' ' \* spaces\_to\_add +
self.source[line\_index])
modified\_lines.append(1 + line\_index) # Line indexed at 1.
line\_index -= 1
return modified\_lines
def fix\_e131(self, result):
"""Fix indentation undistinguish from the next logical line."""
num\_indent\_spaces = int(result['info'].split()[1])
line\_index = result['line'] - 1
target = self.source[line\_index]
spaces\_to\_add = num\_indent\_spaces - len(\_get\_indentation(target))
if spaces\_to\_add >= 0:
self.source[line\_index] = (' ' \* spaces\_to\_add +
self.source[line\_index])
else:
offset = abs(spaces\_to\_add)
self.source[line\_index] = self.source[line\_index][offset:]
def fix\_e201(self, result):
"""Remove extraneous whitespace."""
line\_index = result['line'] - 1
target = self.source[line\_index]
offset = result['column'] - 1
fixed = fix\_whitespace(target,
offset=offset,
replacement='')
self.source[line\_index] = fixed
def fix\_e224(self, result):
"""Remove extraneous whitespace around operator."""
target = self.source[result['line'] - 1]
offset = result['column'] - 1
fixed = target[:offset] + target[offset:].replace('\t', ' ')
self.source[result['line'] - 1] = fixed
def fix\_e225(self, result):
"""Fix missing whitespace around operator."""
target = self.source[result['line'] - 1]
offset = result['column'] - 1
fixed = target[:offset] + ' ' + target[offset:]
# Only proceed if non-whitespace characters match.
# And make sure we don't break the indentation.
if (
fixed.replace(' ', '') == target.replace(' ', '') and
\_get\_indentation(fixed) == \_get\_indentation(target)
):
self.source[result['line'] - 1] = fixed
error\_code = result.get('id', 0)
try:
ts = generate\_tokens(fixed)
except (SyntaxError, tokenize.TokenError):
return
if not check\_syntax(fixed.lstrip()):
return
errors = list(
pycodestyle.missing\_whitespace\_around\_operator(fixed, ts))
for e in reversed(errors):
if error\_code != e[1].split()[0]:
continue
offset = e[0][1]
fixed = fixed[:offset] + ' ' + fixed[offset:]
self.source[result['line'] - 1] = fixed
else:
return []
def fix\_e231(self, result):
"""Add missing whitespace."""
line\_index = result['line'] - 1
target = self.source[line\_index]
offset = result['column']
fixed = target[:offset].rstrip() + ' ' + target[offset:].lstrip()
self.source[line\_index] = fixed
def fix\_e251(self, result):
"""Remove whitespace around parameter '=' sign."""
line\_index = result['line'] - 1
target = self.source[line\_index]
# This is necessary since pycodestyle sometimes reports columns that
# goes past the end of the physical line. This happens in cases like,
# foo(bar\n=None)
c = min(result['column'] - 1,
len(target) - 1)
if target[c].strip():
fixed = target
else:
fixed = target[:c].rstrip() + target[c:].lstrip()
# There could be an escaped newline
#
# def foo(a=\
# 1)
if fixed.endswith(('=\\\n', '=\\\r\n', '=\\\r')):
self.source[line\_index] = fixed.rstrip('\n\r \t\\')
self.source[line\_index + 1] = self.source[line\_index + 1].lstrip()
return [line\_index + 1, line\_index + 2] # Line indexed at 1
self.source[result['line'] - 1] = fixed
def fix\_e262(self, result):
"""Fix spacing after comment hash."""
target = self.source[result['line'] - 1]
offset = result['column']
code = target[:offset].rstrip(' \t#')
comment = target[offset:].lstrip(' \t#')
fixed = code + (' # ' + comment if comment.strip() else '\n')
self.source[result['line'] - 1] = fixed
def fix\_e271(self, result):
"""Fix extraneous whitespace around keywords."""
line\_index = result['line'] - 1
target = self.source[line\_index]
offset = result['column'] - 1
fixed = fix\_whitespace(target,
offset=offset,
replacement=' ')
if fixed == target:
return []
else:
self.source[line\_index] = fixed
def fix\_e301(self, result):
"""Add missing blank line."""
cr = '\n'
self.source[result['line'] - 1] = cr + self.source[result['line'] - 1]
def fix\_e302(self, result):
"""Add missing 2 blank lines."""
add\_linenum = 2 - int(result['info'].split()[-1])
cr = '\n' \* add\_linenum
self.source[result['line'] - 1] = cr + self.source[result['line'] - 1]
def fix\_e303(self, result):
"""Remove extra blank lines."""
delete\_linenum = int(result['info'].split('(')[1].split(')')[0]) - 2
delete\_linenum = max(1, delete\_linenum)
# We need to count because pycodestyle reports an offset line number if
# there are comments.
cnt = 0
line = result['line'] - 2
modified\_lines = []
while cnt < delete\_linenum and line >= 0:
if not self.source[line].strip():
self.source[line] = ''
modified\_lines.append(1 + line) # Line indexed at 1
cnt += 1
line -= 1
return modified\_lines
def fix\_e304(self, result):
"""Remove blank line following function decorator."""
line = result['line'] - 2
if not self.source[line].strip():
self.source[line] = ''
def fix\_e305(self, result):
"""Add missing 2 blank lines after end of function or class."""
add\_delete\_linenum = 2 - int(result['info'].split()[-1])
cnt = 0
offset = result['line'] - 2
modified\_lines = []
if add\_delete\_linenum < 0:
# delete cr
add\_delete\_linenum = abs(add\_delete\_linenum)
while cnt < add\_delete\_linenum and offset >= 0:
if not self.source[offset].strip():
self.source[offset] = ''
modified\_lines.append(1 + offset) # Line indexed at 1
cnt += 1
offset -= 1
else:
# add cr
cr = '\n'
# check comment line
while True:
if offset < 0:
break
line = self.source[offset].lstrip()
if not line:
break
if line[0] != '#':
break
offset -= 1
offset += 1
self.source[offset] = cr + self.source[offset]
modified\_lines.append(1 + offset) # Line indexed at 1.
return modified\_lines
def fix\_e401(self, result):
"""Put imports on separate lines."""
line\_index = result['line'] - 1
target = self.source[line\_index]
offset = result['column'] - 1
if not target.lstrip().startswith('import'):
return []
indentation = re.split(pattern=r'\bimport\b',
string=target, maxsplit=1)[0]
fixed = (target[:offset].rstrip('\t ,') + '\n' +
indentation + 'import ' + target[offset:].lstrip('\t ,'))
self.source[line\_index] = fixed
def fix\_e402(self, result):
(line\_index, offset, target) = get\_index\_offset\_contents(result,
self.source)
for i in range(1, 100):
line = "".join(self.source[line\_index:line\_index+i])
try:
generate\_tokens("".join(line))
except (SyntaxError, tokenize.TokenError):
continue
break
if not (target in self.imports and self.imports[target] != line\_index):
mod\_offset = get\_module\_imports\_on\_top\_of\_file(self.source,
line\_index)
self.source[mod\_offset] = line + self.source[mod\_offset]
for offset in range(i):
self.source[line\_index+offset] = ''
def fix\_long\_line\_logically(self, result, logical):
"""Try to make lines fit within --max-line-length characters."""
if (
not logical or
len(logical[2]) == 1 or
self.source[result['line'] - 1].lstrip().startswith('#')
):
return self.fix\_long\_line\_physically(result)
start\_line\_index = logical[0][0]
end\_line\_index = logical[1][0]
logical\_lines = logical[2]
previous\_line = get\_item(self.source, start\_line\_index - 1, default='')
next\_line = get\_item(self.source, end\_line\_index + 1, default='')
single\_line = join\_logical\_line(''.join(logical\_lines))
try:
fixed = self.fix\_long\_line(
target=single\_line,
previous\_line=previous\_line,
next\_line=next\_line,
original=''.join(logical\_lines))
except (SyntaxError, tokenize.TokenError):
return self.fix\_long\_line\_physically(result)
if fixed:
for line\_index in range(start\_line\_index, end\_line\_index + 1):
self.source[line\_index] = ''
self.source[start\_line\_index] = fixed
return range(start\_line\_index + 1, end\_line\_index + 1)
return []
def fix\_long\_line\_physically(self, result):
"""Try to make lines fit within --max-line-length characters."""
line\_index = result['line'] - 1
target = self.source[line\_index]
previous\_line = get\_item(self.source, line\_index - 1, default='')
next\_line = get\_item(self.source, line\_index + 1, default='')
try:
fixed = self.fix\_long\_line(
target=target,
previous\_line=previous\_line,
next\_line=next\_line,
original=target)
except (SyntaxError, tokenize.TokenError):
return []
if fixed:
self.source[line\_index] = fixed
return [line\_index + 1]
return []
def fix\_long\_line(self, target, previous\_line,
next\_line, original):
cache\_entry = (target, previous\_line, next\_line)
if cache\_entry in self.long\_line\_ignore\_cache:
return []
if target.lstrip().startswith('#'):
if self.options.aggressive:
# Wrap commented lines.
return shorten\_comment(
line=target,
max\_line\_length=self.options.max\_line\_length,
last\_comment=not next\_line.lstrip().startswith('#'))
return []
fixed = get\_fixed\_long\_line(
target=target,
previous\_line=previous\_line,
original=original,
indent\_word=self.indent\_word,
max\_line\_length=self.options.max\_line\_length,
aggressive=self.options.aggressive,
experimental=self.options.experimental,
verbose=self.options.verbose)
if fixed and not code\_almost\_equal(original, fixed):
return fixed
self.long\_line\_ignore\_cache.add(cache\_entry)
return None
def fix\_e502(self, result):
"""Remove extraneous escape of newline."""
(line\_index, \_, target) = get\_index\_offset\_contents(result,
self.source)
self.source[line\_index] = target.rstrip('\n\r \t\\') + '\n'
def fix\_e701(self, result):
"""Put colon-separated compound statement on separate lines."""
line\_index = result['line'] - 1
target = self.source[line\_index]
c = result['column']
fixed\_source = (target[:c] + '\n' +
\_get\_indentation(target) + self.indent\_word +
target[c:].lstrip('\n\r \t\\'))
self.source[result['line'] - 1] = fixed\_source
return [result['line'], result['line'] + 1]
def fix\_e702(self, result, logical):
"""Put semicolon-separated compound statement on separate lines."""
if not logical:
return [] # pragma: no cover
logical\_lines = logical[2]
# Avoid applying this when indented.
# https://docs.python.org/reference/compound\_stmts.html
for line in logical\_lines:
if (result['id'] == 'E702' and ':' in line
and STARTSWITH\_DEF\_REGEX.match(line)):
return []
line\_index = result['line'] - 1
target = self.source[line\_index]
if target.rstrip().endswith('\\'):
# Normalize '1; \\\n2' into '1; 2'.
self.source[line\_index] = target.rstrip('\n \r\t\\')
self.source[line\_index + 1] = self.source[line\_index + 1].lstrip()
return [line\_index + 1, line\_index + 2]
if target.rstrip().endswith(';'):
self.source[line\_index] = target.rstrip('\n \r\t;') + '\n'
return [line\_index + 1]
offset = result['column'] - 1
first = target[:offset].rstrip(';').rstrip()
second = (\_get\_indentation(logical\_lines[0]) +
target[offset:].lstrip(';').lstrip())
# Find inline comment.
inline\_comment = None
if target[offset:].lstrip(';').lstrip()[:2] == '# ':
inline\_comment = target[offset:].lstrip(';')
if inline\_comment:
self.source[line\_index] = first + inline\_comment
else:
self.source[line\_index] = first + '\n' + second
return [line\_index + 1]
def fix\_e704(self, result):
"""Fix multiple statements on one line def"""
(line\_index, \_, target) = get\_index\_offset\_contents(result,
self.source)
match = STARTSWITH\_DEF\_REGEX.match(target)
if match:
self.source[line\_index] = '{}\n{}{}'.format(
match.group(0),
\_get\_indentation(target) + self.indent\_word,
target[match.end(0):].lstrip())
def fix\_e711(self, result):
"""Fix comparison with None."""
(line\_index, offset, target) = get\_index\_offset\_contents(result,
self.source)
right\_offset = offset + 2
if right\_offset >= len(target):
return []
left = target[:offset].rstrip()
center = target[offset:right\_offset]
right = target[right\_offset:].lstrip()
if not right.startswith('None'):
return []
if center.strip() == '==':
new\_center = 'is'
elif center.strip() == '!=':
new\_center = 'is not'
else:
return []
self.source[line\_index] = ' '.join([left, new\_center, right])
def fix\_e712(self, result):
"""Fix (trivial case of) comparison with boolean."""
(line\_index, offset, target) = get\_index\_offset\_contents(result,
self.source)
# Handle very easy "not" special cases.
if re.match(r'^\s\*if [\w."\'\[\]]+ == False:$', target):
self.source[line\_index] = re.sub(r'if ([\w."\'\[\]]+) == False:',
r'if not \1:', target, count=1)
elif re.match(r'^\s\*if [\w."\'\[\]]+ != True:$', target):
self.source[line\_index] = re.sub(r'if ([\w."\'\[\]]+) != True:',
r'if not \1:', target, count=1)
else:
right\_offset = offset + 2
if right\_offset >= len(target):
return []
left = target[:offset].rstrip()
center = target[offset:right\_offset]
right = target[right\_offset:].lstrip()
# Handle simple cases only.
new\_right = None
if center.strip() == '==':
if re.match(r'\bTrue\b', right):
new\_right = re.sub(r'\bTrue\b \*', '', right, count=1)
elif center.strip() == '!=':
if re.match(r'\bFalse\b', right):
new\_right = re.sub(r'\bFalse\b \*', '', right, count=1)
if new\_right is None:
return []
if new\_right[0].isalnum():
new\_right = ' ' + new\_right
self.source[line\_index] = left + new\_right
def fix\_e713(self, result):
"""Fix (trivial case of) non-membership check."""
(line\_index, offset, target) = get\_index\_offset\_contents(result,
self.source)
# to convert once 'not in' -> 'in'
before\_target = target[:offset]
target = target[offset:]
match\_notin = COMPARE\_NEGATIVE\_REGEX\_THROUGH.search(target)
notin\_pos\_start, notin\_pos\_end = 0, 0
if match\_notin:
notin\_pos\_start = match\_notin.start(1)
notin\_pos\_end = match\_notin.end()
target = '{}{} {}'.format(
target[:notin\_pos\_start], 'in', target[notin\_pos\_end:])
# fix 'not in'
match = COMPARE\_NEGATIVE\_REGEX.search(target)
if match:
if match.group(3) == 'in':
pos\_start = match.start(1)
new\_target = '{5}{0}{1} {2} {3} {4}'.format(
target[:pos\_start], match.group(2), match.group(1),
match.group(3), target[match.end():], before\_target)
if match\_notin:
# revert 'in' -> 'not in'
pos\_start = notin\_pos\_start + offset
pos\_end = notin\_pos\_end + offset - 4 # len('not ')
new\_target = '{}{} {}'.format(
new\_target[:pos\_start], 'not in', new\_target[pos\_end:])
self.source[line\_index] = new\_target
def fix\_e714(self, result):
"""Fix object identity should be 'is not' case."""
(line\_index, offset, target) = get\_index\_offset\_contents(result,
self.source)
# to convert once 'is not' -> 'is'
before\_target = target[:offset]
target = target[offset:]
match\_isnot = COMPARE\_NEGATIVE\_REGEX\_THROUGH.search(target)
isnot\_pos\_start, isnot\_pos\_end = 0, 0
if match\_isnot:
isnot\_pos\_start = match\_isnot.start(1)
isnot\_pos\_end = match\_isnot.end()
target = '{}{} {}'.format(
target[:isnot\_pos\_start], 'in', target[isnot\_pos\_end:])
match = COMPARE\_NEGATIVE\_REGEX.search(target)
if match:
if match.group(3).startswith('is'):
pos\_start = match.start(1)
new\_target = '{5}{0}{1} {2} {3} {4}'.format(
target[:pos\_start], match.group(2), match.group(3),
match.group(1), target[match.end():], before\_target)
if match\_isnot:
# revert 'is' -> 'is not'
pos\_start = isnot\_pos\_start + offset
pos\_end = isnot\_pos\_end + offset - 4 # len('not ')
new\_target = '{}{} {}'.format(
new\_target[:pos\_start], 'is not', new\_target[pos\_end:])
self.source[line\_index] = new\_target
def fix\_e722(self, result):
"""fix bare except"""
(line\_index, \_, target) = get\_index\_offset\_contents(result,
self.source)
match = BARE\_EXCEPT\_REGEX.search(target)
if match:
self.source[line\_index] = '{}{}{}'.format(
target[:result['column'] - 1], "except BaseException:",
target[match.end():])
def fix\_e731(self, result):
"""Fix do not assign a lambda expression check."""
(line\_index, \_, target) = get\_index\_offset\_contents(result,
self.source)
match = LAMBDA\_REGEX.search(target)
if match:
end = match.end()
self.source[line\_index] = '{}def {}({}): return {}'.format(
target[:match.start(0)], match.group(1), match.group(2),
target[end:].lstrip())
def fix\_w291(self, result):
"""Remove trailing whitespace."""
fixed\_line = self.source[result['line'] - 1].rstrip()
self.source[result['line'] - 1] = fixed\_line + '\n'
def fix\_w391(self, \_):
"""Remove trailing blank lines."""
blank\_count = 0
for line in reversed(self.source):
line = line.rstrip()
if line:
break
else:
blank\_count += 1
original\_length = len(self.source)
self.source = self.source[:original\_length - blank\_count]
return range(1, 1 + original\_length)
def fix\_w503(self, result):
(line\_index, \_, target) = get\_index\_offset\_contents(result,
self.source)
one\_string\_token = target.split()[0]
try:
ts = generate\_tokens(one\_string\_token)
except (SyntaxError, tokenize.TokenError):
return
if not \_is\_binary\_operator(ts[0][0], one\_string\_token):
return
# find comment
comment\_index = 0
found\_not\_comment\_only\_line = False
comment\_only\_linenum = 0
for i in range(5):
# NOTE: try to parse code in 5 times
if (line\_index - i) < 0:
break
from\_index = line\_index - i - 1
if from\_index < 0 or len(self.source) <= from\_index:
break
to\_index = line\_index + 1
strip\_line = self.source[from\_index].lstrip()
if (
not found\_not\_comment\_only\_line and
strip\_line and strip\_line[0] == '#'
):
comment\_only\_linenum += 1
continue
found\_not\_comment\_only\_line = True
try:
ts = generate\_tokens("".join(self.source[from\_index:to\_index]))
except (SyntaxError, tokenize.TokenError):
continue
newline\_count = 0
newline\_index = []
for index, t in enumerate(ts):
if t[0] in (tokenize.NEWLINE, tokenize.NL):
newline\_index.append(index)
newline\_count += 1
if newline\_count > 2:
tts = ts[newline\_index[-3]:]
else:
tts = ts
old = None
for t in tts:
if t[0] in (tokenize.NEWLINE, tokenize.NL):
newline\_count -= 1
if newline\_count <= 1:
break
if tokenize.COMMENT == t[0] and old and old[0] != tokenize.NL:
comment\_index = old[3][1]
break
old = t
break
i = target.index(one\_string\_token)
fix\_target\_line = line\_index - 1 - comment\_only\_linenum
self.source[line\_index] = '{}{}'.format(
target[:i], target[i + len(one\_string\_token):].lstrip())
nl = find\_newline(self.source[fix\_target\_line:line\_index])
before\_line = self.source[fix\_target\_line]
bl = before\_line.index(nl)
if comment\_index:
self.source[fix\_target\_line] = '{} {} {}'.format(
before\_line[:comment\_index], one\_string\_token,
before\_line[comment\_index + 1:])
else:
self.source[fix\_target\_line] = '{} {}{}'.format(
before\_line[:bl], one\_string\_token, before\_line[bl:])
def fix\_w504(self, result):
(line\_index, \_, target) = get\_index\_offset\_contents(result,
self.source)
# NOTE: is not collect pointed out in pycodestyle==2.4.0
comment\_index = 0
operator\_position = None # (start\_position, end\_position)
for i in range(1, 6):
to\_index = line\_index + i
try:
ts = generate\_tokens("".join(self.source[line\_index:to\_index]))
except (SyntaxError, tokenize.TokenError):
continue
newline\_count = 0
newline\_index = []
for index, t in enumerate(ts):
if \_is\_binary\_operator(t[0], t[1]):
if t[2][0] == 1 and t[3][0] == 1:
operator\_position = (t[2][1], t[3][1])
elif t[0] == tokenize.NAME and t[1] in ("and", "or"):
if t[2][0] == 1 and t[3][0] == 1:
operator\_position = (t[2][1], t[3][1])
elif t[0] in (tokenize.NEWLINE, tokenize.NL):
newline\_index.append(index)
newline\_count += 1
if newline\_count > 2:
tts = ts[:newline\_index[-3]]
else:
tts = ts
old = []
for t in tts:
if tokenize.COMMENT == t[0] and old:
comment\_index = old[3][1]
break
old = t
break
if not operator\_position:
return
target\_operator = target[operator\_position[0]:operator\_position[1]]
if comment\_index:
self.source[line\_index] = '{}{}'.format(
target[:operator\_position[0]].rstrip(),
target[comment\_index:])
else:
self.source[line\_index] = '{}{}{}'.format(
target[:operator\_position[0]].rstrip(),
target[operator\_position[1]:].lstrip(),
target[operator\_position[1]:])
next\_line = self.source[line\_index + 1]
next\_line\_indent = 0
m = re.match(r'\s\*', next\_line)
if m:
next\_line\_indent = m.span()[1]
self.source[line\_index + 1] = '{}{} {}'.format(
next\_line[:next\_line\_indent], target\_operator,
next\_line[next\_line\_indent:])
def fix\_w605(self, result):
(line\_index, \_, target) = get\_index\_offset\_contents(result,
self.source)
try:
tokens = list(generate\_tokens(target))
except (SyntaxError, tokenize.TokenError):
return
for (pos, \_msg) in get\_w605\_position(tokens):
self.source[line\_index] = '{}r{}'.format(
target[:pos], target[pos:])
def get\_w605\_position(tokens):
"""workaround get pointing out position by W605."""
# TODO: When this PR(\*) change is released, use pos of pycodestyle
# \*: https://github.com/PyCQA/pycodestyle/pull/747
valid = [
'\n', '\\', '\'', '"', 'a', 'b', 'f', 'n', 'r', 't', 'v',
'0', '1', '2', '3', '4', '5', '6', '7', 'x',
# Escape sequences only recognized in string literals
'N', 'u', 'U',
]
for token\_type, text, start\_pos, end\_pos, line in tokens:
if token\_type == tokenize.STRING:
quote = text[-3:] if text[-3:] in ('"""', "'''") else text[-1]
# Extract string modifiers (e.g. u or r)
quote\_pos = text.index(quote)
prefix = text[:quote\_pos].lower()
start = quote\_pos + len(quote)
string = text[start:-len(quote)]
if 'r' not in prefix:
pos = string.find('\\')
while pos >= 0:
pos += 1
if string[pos] not in valid:
yield (
# No need to search line, token stores position
start\_pos[1],
"W605 invalid escape sequence '\\%s'" %
string[pos],
)
pos = string.find('\\', pos + 1)
def get\_module\_imports\_on\_top\_of\_file(source, import\_line\_index):
"""return import or from keyword position
example:
> 0: import sys
1: import os
2:
3: def function():
"""
def is\_string\_literal(line):
if line[0] in 'uUbB':
line = line[1:]
if line and line[0] in 'rR':
line = line[1:]
return line and (line[0] == '"' or line[0] == "'")
allowed\_try\_keywords = ('try', 'except', 'else', 'finally')
for cnt, line in enumerate(source):
if not line.rstrip():
continue
elif line.startswith('#'):
continue
if line.startswith('import ') or line.startswith('from '):
if cnt == import\_line\_index:
continue
return cnt
elif pycodestyle.DUNDER\_REGEX.match(line):
continue
elif any(line.startswith(kw) for kw in allowed\_try\_keywords):
continue
elif is\_string\_literal(line):
return cnt
else:
return cnt
return 0
def get\_index\_offset\_contents(result, source):
"""Return (line\_index, column\_offset, line\_contents)."""
line\_index = result['line'] - 1
return (line\_index,
result['column'] - 1,
source[line\_index])
def get\_fixed\_long\_line(target, previous\_line, original,
indent\_word=' ', max\_line\_length=79,
aggressive=False, experimental=False, verbose=False):
"""Break up long line and return result.
Do this by generating multiple reformatted candidates and then
ranking the candidates to heuristically select the best option.
"""
indent = \_get\_indentation(target)
source = target[len(indent):]
assert source.lstrip() == source
assert not target.lstrip().startswith('#')
# Check for partial multiline.
tokens = list(generate\_tokens(source))
candidates = shorten\_line(
tokens, source, indent,
indent\_word,
max\_line\_length,
aggressive=aggressive,
experimental=experimental,
previous\_line=previous\_line)
# Also sort alphabetically as a tie breaker (for determinism).
candidates = sorted(
sorted(set(candidates).union([target, original])),
key=lambda x: line\_shortening\_rank(
x,
indent\_word,
max\_line\_length,
experimental=experimental))
if verbose >= 4:
print(('-' \* 79 + '\n').join([''] + candidates + ['']),
file=wrap\_output(sys.stderr, 'utf-8'))
if candidates:
best\_candidate = candidates[0]
# Don't allow things to get longer.
if longest\_line\_length(best\_candidate) > longest\_line\_length(original):
return None
return best\_candidate
def longest\_line\_length(code):
"""Return length of longest line."""
return max(len(line) for line in code.splitlines())
def join\_logical\_line(logical\_line):
"""Return single line based on logical line input."""
indentation = \_get\_indentation(logical\_line)
return indentation + untokenize\_without\_newlines(
generate\_tokens(logical\_line.lstrip())) + '\n'
def untokenize\_without\_newlines(tokens):
"""Return source code based on tokens."""
text = ''
last\_row = 0
last\_column = -1
for t in tokens:
token\_string = t[1]
(start\_row, start\_column) = t[2]
(end\_row, end\_column) = t[3]
if start\_row > last\_row:
last\_column = 0
if (
(start\_column > last\_column or token\_string == '\n') and
not text.endswith(' ')
):
text += ' '
if token\_string != '\n':
text += token\_string
last\_row = end\_row
last\_column = end\_column
return text.rstrip()
def \_find\_logical(source\_lines):
# Make a variable which is the index of all the starts of lines.
logical\_start = []
logical\_end = []
last\_newline = True
parens = 0
for t in generate\_tokens(''.join(source\_lines)):
if t[0] in [tokenize.COMMENT, tokenize.DEDENT,
tokenize.INDENT, tokenize.NL,
tokenize.ENDMARKER]:
continue
if not parens and t[0] in [tokenize.NEWLINE, tokenize.SEMI]:
last\_newline = True
logical\_end.append((t[3][0] - 1, t[2][1]))
continue
if last\_newline and not parens:
logical\_start.append((t[2][0] - 1, t[2][1]))
last\_newline = False
if t[0] == tokenize.OP:
if t[1] in '([{':
parens += 1
elif t[1] in '}])':
parens -= 1
return (logical\_start, logical\_end)
def \_get\_logical(source\_lines, result, logical\_start, logical\_end):
"""Return the logical line corresponding to the result.
Assumes input is already E702-clean.
"""
row = result['line'] - 1
col = result['column'] - 1
ls = None
le = None
for i in range(0, len(logical\_start), 1):
assert logical\_end
x = logical\_end[i]
if x[0] > row or (x[0] == row and x[1] > col):
le = x
ls = logical\_start[i]
break
if ls is None:
return None
original = source\_lines[ls[0]:le[0] + 1]
return ls, le, original
def get\_item(items, index, default=None):
if 0 <= index < len(items):
return items[index]
return default
def reindent(source, indent\_size):
"""Reindent all lines."""
reindenter = Reindenter(source)
return reindenter.run(indent\_size)
def code\_almost\_equal(a, b):
"""Return True if code is similar.
Ignore whitespace when comparing specific line.
"""
split\_a = split\_and\_strip\_non\_empty\_lines(a)
split\_b = split\_and\_strip\_non\_empty\_lines(b)
if len(split\_a) != len(split\_b):
return False
for (index, \_) in enumerate(split\_a):
if ''.join(split\_a[index].split()) != ''.join(split\_b[index].split()):
return False
return True
def split\_and\_strip\_non\_empty\_lines(text):
"""Return lines split by newline.
Ignore empty lines.
"""
return [line.strip() for line in text.splitlines() if line.strip()]
def fix\_e265(source, aggressive=False): # pylint: disable=unused-argument
"""Format block comments."""
if '#' not in source:
# Optimization.
return source
ignored\_line\_numbers = multiline\_string\_lines(
source,
include\_docstrings=True) | set(commented\_out\_code\_lines(source))
fixed\_lines = []
sio = io.StringIO(source)
for (line\_number, line) in enumerate(sio.readlines(), start=1):
if (
line.lstrip().startswith('#') and
line\_number not in ignored\_line\_numbers and
not pycodestyle.noqa(line)
):
indentation = \_get\_indentation(line)
line = line.lstrip()
# Normalize beginning if not a shebang.
if len(line) > 1:
pos = next((index for index, c in enumerate(line)
if c != '#'))
if (
# Leave multiple spaces like '# ' alone.
(line[:pos].count('#') > 1 or line[1].isalnum() or
not line[1].isspace()) and
line[1] not in ':!' and
# Leave stylistic outlined blocks alone.
not line.rstrip().endswith('#')
):
line = '# ' + line.lstrip('# \t')
fixed\_lines.append(indentation + line)
else:
fixed\_lines.append(line)
return ''.join(fixed\_lines)
def refactor(source, fixer\_names, ignore=None, filename=''):
"""Return refactored code using lib2to3.
Skip if ignore string is produced in the refactored code.
"""
from lib2to3 import pgen2
try:
new\_text = refactor\_with\_2to3(source,
fixer\_names=fixer\_names,
filename=filename)
except (pgen2.parse.ParseError,
SyntaxError,
UnicodeDecodeError,
UnicodeEncodeError):
return source
if ignore:
if ignore in new\_text and ignore not in source:
return source
return new\_text
def code\_to\_2to3(select, ignore, where='', verbose=False):
fixes = set()
for code, fix in CODE\_TO\_2TO3.items():
if code\_match(code, select=select, ignore=ignore):
if verbose:
print('---> Applying {} fix for {}'.format(where,
code.upper()),
file=sys.stderr)
fixes |= set(fix)
return fixes
def fix\_2to3(source,
aggressive=True, select=None, ignore=None, filename='',
where='global', verbose=False):
"""Fix various deprecated code (via lib2to3)."""
if not aggressive:
return source
select = select or []
ignore = ignore or []
return refactor(source,
code\_to\_2to3(select=select,
ignore=ignore,
where=where,
verbose=verbose),
filename=filename)
def fix\_w602(source, aggressive=True):
"""Fix deprecated form of raising exception."""
if not aggressive:
return source
return refactor(source, ['raise'], ignore='with\_traceback')
def find\_newline(source):
"""Return type of newline used in source.
Input is a list of lines.
"""
assert not isinstance(source, unicode)
counter = collections.defaultdict(int)
for line in source:
if line.endswith(CRLF):
counter[CRLF] += 1
elif line.endswith(CR):
counter[CR] += 1
elif line.endswith(LF):
counter[LF] += 1
return (sorted(counter, key=counter.get, reverse=True) or [LF])[0]
def \_get\_indentword(source):
"""Return indentation type."""
indent\_word = ' ' # Default in case source has no indentation
try:
for t in generate\_tokens(source):
if t[0] == token.INDENT:
indent\_word = t[1]
break
except (SyntaxError, tokenize.TokenError):
pass
return indent\_word
def \_get\_indentation(line):
"""Return leading whitespace."""
if line.strip():
non\_whitespace\_index = len(line) - len(line.lstrip())
return line[:non\_whitespace\_index]
return ''
def get\_diff\_text(old, new, filename):
"""Return text of unified diff between old and new."""
newline = '\n'
diff = difflib.unified\_diff(
old, new,
'original/' + filename,
'fixed/' + filename,
lineterm=newline)
text = ''
for line in diff:
text += line
# Work around missing newline (http://bugs.python.org/issue2142).
if text and not line.endswith(newline):
text += newline + r'\ No newline at end of file' + newline
return text
def \_priority\_key(pep8\_result):
"""Key for sorting PEP8 results.
Global fixes should be done first. This is important for things like
indentation.
"""
priority = [
# Fix multiline colon-based before semicolon based.
'e701',
# Break multiline statements early.
'e702',
# Things that make lines longer.
'e225', 'e231',
# Remove extraneous whitespace before breaking lines.
'e201',
# Shorten whitespace in comment before resorting to wrapping.
'e262'
]
middle\_index = 10000
lowest\_priority = [
# We need to shorten lines last since the logical fixer can get in a
# loop, which causes us to exit early.
'e501',
]
key = pep8\_result['id'].lower()
try:
return priority.index(key)
except ValueError:
try:
return middle\_index + lowest\_priority.index(key) + 1
except ValueError:
return middle\_index
def shorten\_line(tokens, source, indentation, indent\_word, max\_line\_length,
aggressive=False, experimental=False, previous\_line=''):
"""Separate line at OPERATOR.
Multiple candidates will be yielded.
"""
for candidate in \_shorten\_line(tokens=tokens,
source=source,
indentation=indentation,
indent\_word=indent\_word,
aggressive=aggressive,
previous\_line=previous\_line):
yield candidate
if aggressive:
for key\_token\_strings in SHORTEN\_OPERATOR\_GROUPS:
shortened = \_shorten\_line\_at\_tokens(
tokens=tokens,
source=source,
indentation=indentation,
indent\_word=indent\_word,
key\_token\_strings=key\_token\_strings,
aggressive=aggressive)
if shortened is not None and shortened != source:
yield shortened
if experimental:
for shortened in \_shorten\_line\_at\_tokens\_new(
tokens=tokens,
source=source,
indentation=indentation,
max\_line\_length=max\_line\_length):
yield shortened
def \_shorten\_line(tokens, source, indentation, indent\_word,
aggressive=False, previous\_line=''):
"""Separate line at OPERATOR.
The input is expected to be free of newlines except for inside multiline
strings and at the end.
Multiple candidates will be yielded.
"""
for (token\_type,
token\_string,
start\_offset,
end\_offset) in token\_offsets(tokens):
if (
token\_type == tokenize.COMMENT and
not is\_probably\_part\_of\_multiline(previous\_line) and
not is\_probably\_part\_of\_multiline(source) and
not source[start\_offset + 1:].strip().lower().startswith(
('noqa', 'pragma:', 'pylint:'))
):
# Move inline comments to previous line.
first = source[:start\_offset]
second = source[start\_offset:]
yield (indentation + second.strip() + '\n' +
indentation + first.strip() + '\n')
elif token\_type == token.OP and token\_string != '=':
# Don't break on '=' after keyword as this violates PEP 8.
assert token\_type != token.INDENT
first = source[:end\_offset]
second\_indent = indentation
if (first.rstrip().endswith('(') and
source[end\_offset:].lstrip().startswith(')')):
pass
elif first.rstrip().endswith('('):
second\_indent += indent\_word
elif '(' in first:
second\_indent += ' ' \* (1 + first.find('('))
else:
second\_indent += indent\_word
second = (second\_indent + source[end\_offset:].lstrip())
if (
not second.strip() or
second.lstrip().startswith('#')
):
continue
# Do not begin a line with a comma
if second.lstrip().startswith(','):
continue
# Do end a line with a dot
if first.rstrip().endswith('.'):
continue
if token\_string in '+-\*/':
fixed = first + ' \\' + '\n' + second
else:
fixed = first + '\n' + second
# Only fix if syntax is okay.
if check\_syntax(normalize\_multiline(fixed)
if aggressive else fixed):
yield indentation + fixed
def \_is\_binary\_operator(token\_type, text):
return ((token\_type == tokenize.OP or text in ['and', 'or']) and
text not in '()[]{},:.;@=%~')
# A convenient way to handle tokens.
Token = collections.namedtuple('Token', ['token\_type', 'token\_string',
'spos', 'epos', 'line'])
class ReformattedLines(object):
"""The reflowed lines of atoms.
Each part of the line is represented as an "atom." They can be moved
around when need be to get the optimal formatting.
"""
###########################################################################
# Private Classes
class \_Indent(object):
"""Represent an indentation in the atom stream."""
def \_\_init\_\_(self, indent\_amt):
self.\_indent\_amt = indent\_amt
def emit(self):
return ' ' \* self.\_indent\_amt
@property
def size(self):
return self.\_indent\_amt
class \_Space(object):
"""Represent a space in the atom stream."""
def emit(self):
return ' '
@property
def size(self):
return 1
class \_LineBreak(object):
"""Represent a line break in the atom stream."""
def emit(self):
return '\n'
@property
def size(self):
return 0
def \_\_init\_\_(self, max\_line\_length):
self.\_max\_line\_length = max\_line\_length
self.\_lines = []
self.\_bracket\_depth = 0
self.\_prev\_item = None
self.\_prev\_prev\_item = None
def \_\_repr\_\_(self):
return self.emit()
###########################################################################
# Public Methods
def add(self, obj, indent\_amt, break\_after\_open\_bracket):
if isinstance(obj, Atom):
self.\_add\_item(obj, indent\_amt)
return
self.\_add\_container(obj, indent\_amt, break\_after\_open\_bracket)
def add\_comment(self, item):
num\_spaces = 2
if len(self.\_lines) > 1:
if isinstance(self.\_lines[-1], self.\_Space):
num\_spaces -= 1
if len(self.\_lines) > 2:
if isinstance(self.\_lines[-2], self.\_Space):
num\_spaces -= 1
while num\_spaces > 0:
self.\_lines.append(self.\_Space())
num\_spaces -= 1
self.\_lines.append(item)
def add\_indent(self, indent\_amt):
self.\_lines.append(self.\_Indent(indent\_amt))
def add\_line\_break(self, indent):
self.\_lines.append(self.\_LineBreak())
self.add\_indent(len(indent))
def add\_line\_break\_at(self, index, indent\_amt):
self.\_lines.insert(index, self.\_LineBreak())
self.\_lines.insert(index + 1, self.\_Indent(indent\_amt))
def add\_space\_if\_needed(self, curr\_text, equal=False):
if (
not self.\_lines or isinstance(
self.\_lines[-1], (self.\_LineBreak, self.\_Indent, self.\_Space))
):
return
prev\_text = unicode(self.\_prev\_item)
prev\_prev\_text = (
unicode(self.\_prev\_prev\_item) if self.\_prev\_prev\_item else '')
if (
# The previous item was a keyword or identifier and the current
# item isn't an operator that doesn't require a space.
((self.\_prev\_item.is\_keyword or self.\_prev\_item.is\_string or
self.\_prev\_item.is\_name or self.\_prev\_item.is\_number) and
(curr\_text[0] not in '([{.,:}])' or
(curr\_text[0] == '=' and equal))) or
# Don't place spaces around a '.', unless it's in an 'import'
# statement.
((prev\_prev\_text != 'from' and prev\_text[-1] != '.' and
curr\_text != 'import') and
# Don't place a space before a colon.
curr\_text[0] != ':' and
# Don't split up ending brackets by spaces.
((prev\_text[-1] in '}])' and curr\_text[0] not in '.,}])') or
# Put a space after a colon or comma.
prev\_text[-1] in ':,' or
# Put space around '=' if asked to.
(equal and prev\_text == '=') or
# Put spaces around non-unary arithmetic operators.
((self.\_prev\_prev\_item and
(prev\_text not in '+-' and
(self.\_prev\_prev\_item.is\_name or
self.\_prev\_prev\_item.is\_number or
self.\_prev\_prev\_item.is\_string)) and
prev\_text in ('+', '-', '%', '\*', '/', '//', '\*\*', 'in')))))
):
self.\_lines.append(self.\_Space())
def previous\_item(self):
"""Return the previous non-whitespace item."""
return self.\_prev\_item
def fits\_on\_current\_line(self, item\_extent):
return self.current\_size() + item\_extent <= self.\_max\_line\_length
def current\_size(self):
"""The size of the current line minus the indentation."""
size = 0
for item in reversed(self.\_lines):
size += item.size
if isinstance(item, self.\_LineBreak):
break
return size
def line\_empty(self):
return (self.\_lines and
isinstance(self.\_lines[-1],
(self.\_LineBreak, self.\_Indent)))
def emit(self):
string = ''
for item in self.\_lines:
if isinstance(item, self.\_LineBreak):
string = string.rstrip()
string += item.emit()
return string.rstrip() + '\n'
###########################################################################
# Private Methods
def \_add\_item(self, item, indent\_amt):
"""Add an item to the line.
Reflow the line to get the best formatting after the item is
inserted. The bracket depth indicates if the item is being
inserted inside of a container or not.
"""
if self.\_prev\_item and self.\_prev\_item.is\_string and item.is\_string:
# Place consecutive string literals on separate lines.
self.\_lines.append(self.\_LineBreak())
self.\_lines.append(self.\_Indent(indent\_amt))
item\_text = unicode(item)
if self.\_lines and self.\_bracket\_depth:
# Adding the item into a container.
self.\_prevent\_default\_initializer\_splitting(item, indent\_amt)
if item\_text in '.,)]}':
self.\_split\_after\_delimiter(item, indent\_amt)
elif self.\_lines and not self.line\_empty():
# Adding the item outside of a container.
if self.fits\_on\_current\_line(len(item\_text)):
self.\_enforce\_space(item)
else:
# Line break for the new item.
self.\_lines.append(self.\_LineBreak())
self.\_lines.append(self.\_Indent(indent\_amt))
self.\_lines.append(item)
self.\_prev\_item, self.\_prev\_prev\_item = item, self.\_prev\_item
if item\_text in '([{':
self.\_bracket\_depth += 1
elif item\_text in '}])':
self.\_bracket\_depth -= 1
assert self.\_bracket\_depth >= 0
def \_add\_container(self, container, indent\_amt, break\_after\_open\_bracket):
actual\_indent = indent\_amt + 1
if (
unicode(self.\_prev\_item) != '=' and
not self.line\_empty() and
not self.fits\_on\_current\_line(
container.size + self.\_bracket\_depth + 2)
):
if unicode(container)[0] == '(' and self.\_prev\_item.is\_name:
# Don't split before the opening bracket of a call.
break\_after\_open\_bracket = True
actual\_indent = indent\_amt + 4
elif (
break\_after\_open\_bracket or
unicode(self.\_prev\_item) not in '([{'
):
# If the container doesn't fit on the current line and the
# current line isn't empty, place the container on the next
# line.
self.\_lines.append(self.\_LineBreak())
self.\_lines.append(self.\_Indent(indent\_amt))
break\_after\_open\_bracket = False
else:
actual\_indent = self.current\_size() + 1
break\_after\_open\_bracket = False
if isinstance(container, (ListComprehension, IfExpression)):
actual\_indent = indent\_amt
# Increase the continued indentation only if recursing on a
# container.
container.reflow(self, ' ' \* actual\_indent,
break\_after\_open\_bracket=break\_after\_open\_bracket)
def \_prevent\_default\_initializer\_splitting(self, item, indent\_amt):
"""Prevent splitting between a default initializer.
When there is a default initializer, it's best to keep it all on
the same line. It's nicer and more readable, even if it goes
over the maximum allowable line length. This goes back along the
current line to determine if we have a default initializer, and,
if so, to remove extraneous whitespaces and add a line
break/indent before it if needed.
"""
if unicode(item) == '=':
# This is the assignment in the initializer. Just remove spaces for
# now.
self.\_delete\_whitespace()
return
if (not self.\_prev\_item or not self.\_prev\_prev\_item or
unicode(self.\_prev\_item) != '='):
return
self.\_delete\_whitespace()
prev\_prev\_index = self.\_lines.index(self.\_prev\_prev\_item)
if (
isinstance(self.\_lines[prev\_prev\_index - 1], self.\_Indent) or
self.fits\_on\_current\_line(item.size + 1)
):
# The default initializer is already the only item on this line.
# Don't insert a newline here.
return
# Replace the space with a newline/indent combo.
if isinstance(self.\_lines[prev\_prev\_index - 1], self.\_Space):
del self.\_lines[prev\_prev\_index - 1]
self.add\_line\_break\_at(self.\_lines.index(self.\_prev\_prev\_item),
indent\_amt)
def \_split\_after\_delimiter(self, item, indent\_amt):
"""Split the line only after a delimiter."""
self.\_delete\_whitespace()
if self.fits\_on\_current\_line(item.size):
return
last\_space = None
for current\_item in reversed(self.\_lines):
if (
last\_space and
(not isinstance(current\_item, Atom) or
not current\_item.is\_colon)
):
break
else:
last\_space = None
if isinstance(current\_item, self.\_Space):
last\_space = current\_item
if isinstance(current\_item, (self.\_LineBreak, self.\_Indent)):
return
if not last\_space:
return
self.add\_line\_break\_at(self.\_lines.index(last\_space), indent\_amt)
def \_enforce\_space(self, item):
"""Enforce a space in certain situations.
There are cases where we will want a space where normally we
wouldn't put one. This just enforces the addition of a space.
"""
if isinstance(self.\_lines[-1],
(self.\_Space, self.\_LineBreak, self.\_Indent)):
return
if not self.\_prev\_item:
return
item\_text = unicode(item)
prev\_text = unicode(self.\_prev\_item)
# Prefer a space around a '.' in an import statement, and between the
# 'import' and '('.
if (
(item\_text == '.' and prev\_text == 'from') or
(item\_text == 'import' and prev\_text == '.') or
(item\_text == '(' and prev\_text == 'import')
):
self.\_lines.append(self.\_Space())
def \_delete\_whitespace(self):
"""Delete all whitespace from the end of the line."""
while isinstance(self.\_lines[-1], (self.\_Space, self.\_LineBreak,
self.\_Indent)):
del self.\_lines[-1]
class Atom(object):
"""The smallest unbreakable unit that can be reflowed."""
def \_\_init\_\_(self, atom):
self.\_atom = atom
def \_\_repr\_\_(self):
return self.\_atom.token\_string
def \_\_len\_\_(self):
return self.size
def reflow(
self, reflowed\_lines, continued\_indent, extent,
break\_after\_open\_bracket=False,
is\_list\_comp\_or\_if\_expr=False,
next\_is\_dot=False
):
if self.\_atom.token\_type == tokenize.COMMENT:
reflowed\_lines.add\_comment(self)
return
total\_size = extent if extent else self.size
if self.\_atom.token\_string not in ',:([{}])':
# Some atoms will need an extra 1-sized space token after them.
total\_size += 1
prev\_item = reflowed\_lines.previous\_item()
if (
not is\_list\_comp\_or\_if\_expr and
not reflowed\_lines.fits\_on\_current\_line(total\_size) and
not (next\_is\_dot and
reflowed\_lines.fits\_on\_current\_line(self.size + 1)) and
not reflowed\_lines.line\_empty() and
not self.is\_colon and
not (prev\_item and prev\_item.is\_name and
unicode(self) == '(')
):
# Start a new line if there is already something on the line and
# adding this atom would make it go over the max line length.
reflowed\_lines.add\_line\_break(continued\_indent)
else:
reflowed\_lines.add\_space\_if\_needed(unicode(self))
reflowed\_lines.add(self, len(continued\_indent),
break\_after\_open\_bracket)
def emit(self):
return self.\_\_repr\_\_()
@property
def is\_keyword(self):
return keyword.iskeyword(self.\_atom.token\_string)
@property
def is\_string(self):
return self.\_atom.token\_type == tokenize.STRING
@property
def is\_name(self):
return self.\_atom.token\_type == tokenize.NAME
@property
def is\_number(self):
return self.\_atom.token\_type == tokenize.NUMBER
@property
def is\_comma(self):
return self.\_atom.token\_string == ','
@property
def is\_colon(self):
return self.\_atom.token\_string == ':'
@property
def size(self):
return len(self.\_atom.token\_string)
class Container(object):
"""Base class for all container types."""
def \_\_init\_\_(self, items):
self.\_items = items
def \_\_repr\_\_(self):
string = ''
last\_was\_keyword = False
for item in self.\_items:
if item.is\_comma:
string += ', '
elif item.is\_colon:
string += ': '
else:
item\_string = unicode(item)
if (
string and
(last\_was\_keyword or
(not string.endswith(tuple('([{,.:}]) ')) and
not item\_string.startswith(tuple('([{,.:}])'))))
):
string += ' '
string += item\_string
last\_was\_keyword = item.is\_keyword
return string
def \_\_iter\_\_(self):
for element in self.\_items:
yield element
def \_\_getitem\_\_(self, idx):
return self.\_items[idx]
def reflow(self, reflowed\_lines, continued\_indent,
break\_after\_open\_bracket=False):
last\_was\_container = False
for (index, item) in enumerate(self.\_items):
next\_item = get\_item(self.\_items, index + 1)
if isinstance(item, Atom):
is\_list\_comp\_or\_if\_expr = (
isinstance(self, (ListComprehension, IfExpression)))
item.reflow(reflowed\_lines, continued\_indent,
self.\_get\_extent(index),
is\_list\_comp\_or\_if\_expr=is\_list\_comp\_or\_if\_expr,
next\_is\_dot=(next\_item and
unicode(next\_item) == '.'))
if last\_was\_container and item.is\_comma:
reflowed\_lines.add\_line\_break(continued\_indent)
last\_was\_container = False
else: # isinstance(item, Container)
reflowed\_lines.add(item, len(continued\_indent),
break\_after\_open\_bracket)
last\_was\_container = not isinstance(item, (ListComprehension,
IfExpression))
if (
break\_after\_open\_bracket and index == 0 and
# Prefer to keep empty containers together instead of
# separating them.
unicode(item) == self.open\_bracket and
(not next\_item or unicode(next\_item) != self.close\_bracket) and
(len(self.\_items) != 3 or not isinstance(next\_item, Atom))
):
reflowed\_lines.add\_line\_break(continued\_indent)
break\_after\_open\_bracket = False
else:
next\_next\_item = get\_item(self.\_items, index + 2)
if (
unicode(item) not in ['.', '%', 'in'] and
next\_item and not isinstance(next\_item, Container) and
unicode(next\_item) != ':' and
next\_next\_item and (not isinstance(next\_next\_item, Atom) or
unicode(next\_item) == 'not') and
not reflowed\_lines.line\_empty() and
not reflowed\_lines.fits\_on\_current\_line(
self.\_get\_extent(index + 1) + 2)
):
reflowed\_lines.add\_line\_break(continued\_indent)
def \_get\_extent(self, index):
"""The extent of the full element.
E.g., the length of a function call or keyword.
"""
extent = 0
prev\_item = get\_item(self.\_items, index - 1)
seen\_dot = prev\_item and unicode(prev\_item) == '.'
while index < len(self.\_items):
item = get\_item(self.\_items, index)
index += 1
if isinstance(item, (ListComprehension, IfExpression)):
break
if isinstance(item, Container):
if prev\_item and prev\_item.is\_name:
if seen\_dot:
extent += 1
else:
extent += item.size
prev\_item = item
continue
elif (unicode(item) not in ['.', '=', ':', 'not'] and
not item.is\_name and not item.is\_string):
break
if unicode(item) == '.':
seen\_dot = True
extent += item.size
prev\_item = item
return extent
@property
def is\_string(self):
return False
@property
def size(self):
return len(self.\_\_repr\_\_())
@property
def is\_keyword(self):
return False
@property
def is\_name(self):
return False
@property
def is\_comma(self):
return False
@property
def is\_colon(self):
return False
@property
def open\_bracket(self):
return None
@property
def close\_bracket(self):
return None
class Tuple(Container):
"""A high-level representation of a tuple."""
@property
def open\_bracket(self):
return '('
@property
def close\_bracket(self):
return ')'
class List(Container):
"""A high-level representation of a list."""
@property
def open\_bracket(self):
return '['
@property
def close\_bracket(self):
return ']'
class DictOrSet(Container):
"""A high-level representation of a dictionary or set."""
@property
def open\_bracket(self):
return '{'
@property
def close\_bracket(self):
return '}'
class ListComprehension(Container):
"""A high-level representation of a list comprehension."""
@property
def size(self):
length = 0
for item in self.\_items:
if isinstance(item, IfExpression):
break
length += item.size
return length
class IfExpression(Container):
"""A high-level representation of an if-expression."""
def \_parse\_container(tokens, index, for\_or\_if=None):
"""Parse a high-level container, such as a list, tuple, etc."""
# Store the opening bracket.
items = [Atom(Token(\*tokens[index]))]
index += 1
num\_tokens = len(tokens)
while index < num\_tokens:
tok = Token(\*tokens[index])
if tok.token\_string in ',)]}':
# First check if we're at the end of a list comprehension or
# if-expression. Don't add the ending token as part of the list
# comprehension or if-expression, because they aren't part of those
# constructs.
if for\_or\_if == 'for':
return (ListComprehension(items), index - 1)
elif for\_or\_if == 'if':
return (IfExpression(items), index - 1)
# We've reached the end of a container.
items.append(Atom(tok))
# If not, then we are at the end of a container.
if tok.token\_string == ')':
# The end of a tuple.
return (Tuple(items), index)
elif tok.token\_string == ']':
# The end of a list.
return (List(items), index)
elif tok.token\_string == '}':
# The end of a dictionary or set.
return (DictOrSet(items), index)
elif tok.token\_string in '([{':
# A sub-container is being defined.
(container, index) = \_parse\_container(tokens, index)
items.append(container)
elif tok.token\_string == 'for':
(container, index) = \_parse\_container(tokens, index, 'for')
items.append(container)
elif tok.token\_string == 'if':
(container, index) = \_parse\_container(tokens, index, 'if')
items.append(container)
else:
items.append(Atom(tok))
index += 1
return (None, None)
def \_parse\_tokens(tokens):
"""Parse the tokens.
This converts the tokens into a form where we can manipulate them
more easily.
"""
index = 0
parsed\_tokens = []
num\_tokens = len(tokens)
while index < num\_tokens:
tok = Token(\*tokens[index])
assert tok.token\_type != token.INDENT
if tok.token\_type == tokenize.NEWLINE:
# There's only one newline and it's at the end.
break
if tok.token\_string in '([{':
(container, index) = \_parse\_container(tokens, index)
if not container:
return None
parsed\_tokens.append(container)
else:
parsed\_tokens.append(Atom(tok))
index += 1
return parsed\_tokens
def \_reflow\_lines(parsed\_tokens, indentation, max\_line\_length,
start\_on\_prefix\_line):
"""Reflow the lines so that it looks nice."""
if unicode(parsed\_tokens[0]) == 'def':
# A function definition gets indented a bit more.
continued\_indent = indentation + ' ' \* 2 \* DEFAULT\_INDENT\_SIZE
else:
continued\_indent = indentation + ' ' \* DEFAULT\_INDENT\_SIZE
break\_after\_open\_bracket = not start\_on\_prefix\_line
lines = ReformattedLines(max\_line\_length)
lines.add\_indent(len(indentation.lstrip('\r\n')))
if not start\_on\_prefix\_line:
# If splitting after the opening bracket will cause the first element
# to be aligned weirdly, don't try it.
first\_token = get\_item(parsed\_tokens, 0)
second\_token = get\_item(parsed\_tokens, 1)
if (
first\_token and second\_token and
unicode(second\_token)[0] == '(' and
len(indentation) + len(first\_token) + 1 == len(continued\_indent)
):
return None
for item in parsed\_tokens:
lines.add\_space\_if\_needed(unicode(item), equal=True)
save\_continued\_indent = continued\_indent
if start\_on\_prefix\_line and isinstance(item, Container):
start\_on\_prefix\_line = False
continued\_indent = ' ' \* (lines.current\_size() + 1)
item.reflow(lines, continued\_indent, break\_after\_open\_bracket)
continued\_indent = save\_continued\_indent
return lines.emit()
def \_shorten\_line\_at\_tokens\_new(tokens, source, indentation,
max\_line\_length):
"""Shorten the line taking its length into account.
The input is expected to be free of newlines except for inside
multiline strings and at the end.
"""
# Yield the original source so to see if it's a better choice than the
# shortened candidate lines we generate here.
yield indentation + source
parsed\_tokens = \_parse\_tokens(tokens)
if parsed\_tokens:
# Perform two reflows. The first one starts on the same line as the
# prefix. The second starts on the line after the prefix.
fixed = \_reflow\_lines(parsed\_tokens, indentation, max\_line\_length,
start\_on\_prefix\_line=True)
if fixed and check\_syntax(normalize\_multiline(fixed.lstrip())):
yield fixed
fixed = \_reflow\_lines(parsed\_tokens, indentation, max\_line\_length,
start\_on\_prefix\_line=False)
if fixed and check\_syntax(normalize\_multiline(fixed.lstrip())):
yield fixed
def \_shorten\_line\_at\_tokens(tokens, source, indentation, indent\_word,
key\_token\_strings, aggressive):
"""Separate line by breaking at tokens in key\_token\_strings.
The input is expected to be free of newlines except for inside
multiline strings and at the end.
"""
offsets = []
for (index, \_t) in enumerate(token\_offsets(tokens)):
(token\_type,
token\_string,
start\_offset,
end\_offset) = \_t
assert token\_type != token.INDENT
if token\_string in key\_token\_strings:
# Do not break in containers with zero or one items.
unwanted\_next\_token = {
'(': ')',
'[': ']',
'{': '}'}.get(token\_string)
if unwanted\_next\_token:
if (
get\_item(tokens,
index + 1,
default=[None, None])[1] == unwanted\_next\_token or
get\_item(tokens,
index + 2,
default=[None, None])[1] == unwanted\_next\_token
):
continue
if (
index > 2 and token\_string == '(' and
tokens[index - 1][1] in ',(%['
):
# Don't split after a tuple start, or before a tuple start if
# the tuple is in a list.
continue
if end\_offset < len(source) - 1:
# Don't split right before newline.
offsets.append(end\_offset)
else:
# Break at adjacent strings. These were probably meant to be on
# separate lines in the first place.
previous\_token = get\_item(tokens, index - 1)
if (
token\_type == tokenize.STRING and
previous\_token and previous\_token[0] == tokenize.STRING
):
offsets.append(start\_offset)
current\_indent = None
fixed = None
for line in split\_at\_offsets(source, offsets):
if fixed:
fixed += '\n' + current\_indent + line
for symbol in '([{':
if line.endswith(symbol):
current\_indent += indent\_word
else:
# First line.
fixed = line
assert not current\_indent
current\_indent = indent\_word
assert fixed is not None
if check\_syntax(normalize\_multiline(fixed)
if aggressive > 1 else fixed):
return indentation + fixed
return None
def token\_offsets(tokens):
"""Yield tokens and offsets."""
end\_offset = 0
previous\_end\_row = 0
previous\_end\_column = 0
for t in tokens:
token\_type = t[0]
token\_string = t[1]
(start\_row, start\_column) = t[2]
(end\_row, end\_column) = t[3]
# Account for the whitespace between tokens.
end\_offset += start\_column
if previous\_end\_row == start\_row:
end\_offset -= previous\_end\_column
# Record the start offset of the token.
start\_offset = end\_offset
# Account for the length of the token itself.
end\_offset += len(token\_string)
yield (token\_type,
token\_string,
start\_offset,
end\_offset)
previous\_end\_row = end\_row
previous\_end\_column = end\_column
def normalize\_multiline(line):
"""Normalize multiline-related code that will cause syntax error.
This is for purposes of checking syntax.
"""
if line.startswith('def ') and line.rstrip().endswith(':'):
return line + ' pass'
elif line.startswith('return '):
return 'def \_(): ' + line
elif line.startswith('@'):
return line + 'def \_(): pass'
elif line.startswith('class '):
return line + ' pass'
elif line.startswith(('if ', 'elif ', 'for ', 'while ')):
return line + ' pass'
return line
def fix\_whitespace(line, offset, replacement):
"""Replace whitespace at offset and return fixed line."""
# Replace escaped newlines too
left = line[:offset].rstrip('\n\r \t\\')
right = line[offset:].lstrip('\n\r \t\\')
if right.startswith('#'):
return line
return left + replacement + right
def \_execute\_pep8(pep8\_options, source):
"""Execute pycodestyle via python method calls."""
class QuietReport(pycodestyle.BaseReport):
"""Version of checker that does not print."""
def \_\_init\_\_(self, options):
super(QuietReport, self).\_\_init\_\_(options)
self.\_\_full\_error\_results = []
def error(self, line\_number, offset, text, check):
"""Collect errors."""
code = super(QuietReport, self).error(line\_number,
offset,
text,
check)
if code:
self.\_\_full\_error\_results.append(
{'id': code,
'line': line\_number,
'column': offset + 1,
'info': text})
def full\_error\_results(self):
"""Return error results in detail.
Results are in the form of a list of dictionaries. Each
dictionary contains 'id', 'line', 'column', and 'info'.
"""
return self.\_\_full\_error\_results
checker = pycodestyle.Checker('', lines=source, reporter=QuietReport,
\*\*pep8\_options)
checker.check\_all()
return checker.report.full\_error\_results()
def \_remove\_leading\_and\_normalize(line):
# ignore FF in first lstrip()
return line.lstrip(' \t\v').rstrip(CR + LF) + '\n'
class Reindenter(object):
"""Reindents badly-indented code to uniformly use four-space indentation.
Released to the public domain, by Tim Peters, 03 October 2000.
"""
def \_\_init\_\_(self, input\_text):
sio = io.StringIO(input\_text)
source\_lines = sio.readlines()
self.string\_content\_line\_numbers = multiline\_string\_lines(input\_text)
# File lines, rstripped & tab-expanded. Dummy at start is so
# that we can use tokenize's 1-based line numbering easily.
# Note that a line is all-blank iff it is a newline.
self.lines = []
for line\_number, line in enumerate(source\_lines, start=1):
# Do not modify if inside a multiline string.
if line\_number in self.string\_content\_line\_numbers:
self.lines.append(line)
else:
# Only expand leading tabs.
self.lines.append(\_get\_indentation(line).expandtabs() +
\_remove\_leading\_and\_normalize(line))
self.lines.insert(0, None)
self.index = 1 # index into self.lines of next line
self.input\_text = input\_text
def run(self, indent\_size=DEFAULT\_INDENT\_SIZE):
"""Fix indentation and return modified line numbers.
Line numbers are indexed at 1.
"""
if indent\_size < 1:
return self.input\_text
try:
stats = \_reindent\_stats(tokenize.generate\_tokens(self.getline))
except (SyntaxError, tokenize.TokenError):
return self.input\_text
# Remove trailing empty lines.
lines = self.lines
# Sentinel.
stats.append((len(lines), 0))
# Map count of leading spaces to # we want.
have2want = {}
# Program after transformation.
after = []
# Copy over initial empty lines -- there's nothing to do until
# we see a line with \*something\* on it.
i = stats[0][0]
after.extend(lines[1:i])
for i in range(len(stats) - 1):
thisstmt, thislevel = stats[i]
nextstmt = stats[i + 1][0]
have = \_leading\_space\_count(lines[thisstmt])
want = thislevel \* indent\_size
if want < 0:
# A comment line.
if have:
# An indented comment line. If we saw the same
# indentation before, reuse what it most recently
# mapped to.
want = have2want.get(have, -1)
if want < 0:
# Then it probably belongs to the next real stmt.
for j in range(i + 1, len(stats) - 1):
jline, jlevel = stats[j]
if jlevel >= 0:
if have == \_leading\_space\_count(lines[jline]):
want = jlevel \* indent\_size
break
if want < 0: # Maybe it's a hanging
# comment like this one,
# in which case we should shift it like its base
# line got shifted.
for j in range(i - 1, -1, -1):
jline, jlevel = stats[j]
if jlevel >= 0:
want = (have + \_leading\_space\_count(
after[jline - 1]) -
\_leading\_space\_count(lines[jline]))
break
if want < 0:
# Still no luck -- leave it alone.
want = have
else:
want = 0
assert want >= 0
have2want[have] = want
diff = want - have
if diff == 0 or have == 0:
after.extend(lines[thisstmt:nextstmt])
else:
for line\_number, line in enumerate(lines[thisstmt:nextstmt],
start=thisstmt):
if line\_number in self.string\_content\_line\_numbers:
after.append(line)
elif diff > 0:
if line == '\n':
after.append(line)
else:
after.append(' ' \* diff + line)
else:
remove = min(\_leading\_space\_count(line), -diff)
after.append(line[remove:])
return ''.join(after)
def getline(self):
"""Line-getter for tokenize."""
if self.index >= len(self.lines):
line = ''
else:
line = self.lines[self.index]
self.index += 1
return line
def \_reindent\_stats(tokens):
"""Return list of (lineno, indentlevel) pairs.
One for each stmt and comment line. indentlevel is -1 for comment
lines, as a signal that tokenize doesn't know what to do about them;
indeed, they're our headache!
"""
find\_stmt = 1 # Next token begins a fresh stmt?
level = 0 # Current indent level.
stats = []
for t in tokens:
token\_type = t[0]
sline = t[2][0]
line = t[4]
if token\_type == tokenize.NEWLINE:
# A program statement, or ENDMARKER, will eventually follow,
# after some (possibly empty) run of tokens of the form
# (NL | COMMENT)\* (INDENT | DEDENT+)?
find\_stmt = 1
elif token\_type == tokenize.INDENT:
find\_stmt = 1
level += 1
elif token\_type == tokenize.DEDENT:
find\_stmt = 1
level -= 1
elif token\_type == tokenize.COMMENT:
if find\_stmt:
stats.append((sline, -1))
# But we're still looking for a new stmt, so leave
# find\_stmt alone.
elif token\_type == tokenize.NL:
pass
elif find\_stmt:
# This is the first "real token" following a NEWLINE, so it
# must be the first token of the next program statement, or an
# ENDMARKER.
find\_stmt = 0
if line: # Not endmarker.
stats.append((sline, level))
return stats
def \_leading\_space\_count(line):
"""Return number of leading spaces in line."""
i = 0
while i < len(line) and line[i] == ' ':
i += 1
return i
def refactor\_with\_2to3(source\_text, fixer\_names, filename=''):
"""Use lib2to3 to refactor the source.
Return the refactored source code.
"""
from lib2to3.refactor import RefactoringTool
fixers = ['lib2to3.fixes.fix\_' + name for name in fixer\_names]
tool = RefactoringTool(fixer\_names=fixers, explicit=fixers)
from lib2to3.pgen2 import tokenize as lib2to3\_tokenize
try:
# The name parameter is necessary particularly for the "import" fixer.
return unicode(tool.refactor\_string(source\_text, name=filename))
except lib2to3\_tokenize.TokenError:
return source\_text
def check\_syntax(code):
"""Return True if syntax is okay."""
try:
return compile(code, '<string>', 'exec', dont\_inherit=True)
except (SyntaxError, TypeError, ValueError):
return False
def filter\_results(source, results, aggressive):
"""Filter out spurious reports from pycodestyle.
If aggressive is True, we allow possibly unsafe fixes (E711, E712).
"""
non\_docstring\_string\_line\_numbers = multiline\_string\_lines(
source, include\_docstrings=False)
all\_string\_line\_numbers = multiline\_string\_lines(
source, include\_docstrings=True)
commented\_out\_code\_line\_numbers = commented\_out\_code\_lines(source)
has\_e901 = any(result['id'].lower() == 'e901' for result in results)
for r in results:
issue\_id = r['id'].lower()
if r['line'] in non\_docstring\_string\_line\_numbers:
if issue\_id.startswith(('e1', 'e501', 'w191')):
continue
if r['line'] in all\_string\_line\_numbers:
if issue\_id in ['e501']:
continue
# We must offset by 1 for lines that contain the trailing contents of
# multiline strings.
if not aggressive and (r['line'] + 1) in all\_string\_line\_numbers:
# Do not modify multiline strings in non-aggressive mode. Remove
# trailing whitespace could break doctests.
if issue\_id.startswith(('w29', 'w39')):
continue
if aggressive <= 0:
if issue\_id.startswith(('e711', 'e72', 'w6')):
continue
if aggressive <= 1:
if issue\_id.startswith(('e712', 'e713', 'e714')):
continue
if aggressive <= 2:
if issue\_id.startswith(('e704')):
continue
if r['line'] in commented\_out\_code\_line\_numbers:
if issue\_id.startswith(('e26', 'e501')):
continue
# Do not touch indentation if there is a token error caused by
# incomplete multi-line statement. Otherwise, we risk screwing up the
# indentation.
if has\_e901:
if issue\_id.startswith(('e1', 'e7')):
continue
yield r
def multiline\_string\_lines(source, include\_docstrings=False):
"""Return line numbers that are within multiline strings.
The line numbers are indexed at 1.
Docstrings are ignored.
"""
line\_numbers = set()
previous\_token\_type = ''
try:
for t in generate\_tokens(source):
token\_type = t[0]
start\_row = t[2][0]
end\_row = t[3][0]
if token\_type == tokenize.STRING and start\_row != end\_row:
if (
include\_docstrings or
previous\_token\_type != tokenize.INDENT
):
# We increment by one since we want the contents of the
# string.
line\_numbers |= set(range(1 + start\_row, 1 + end\_row))
previous\_token\_type = token\_type
except (SyntaxError, tokenize.TokenError):
pass
return line\_numbers
def commented\_out\_code\_lines(source):
"""Return line numbers of comments that are likely code.
Commented-out code is bad practice, but modifying it just adds even
more clutter.
"""
line\_numbers = []
try:
for t in generate\_tokens(source):
token\_type = t[0]
token\_string = t[1]
start\_row = t[2][0]
line = t[4]
# Ignore inline comments.
if not line.lstrip().startswith('#'):
continue
if token\_type == tokenize.COMMENT:
stripped\_line = token\_string.lstrip('#').strip()
if (
' ' in stripped\_line and
'#' not in stripped\_line and
check\_syntax(stripped\_line)
):
line\_numbers.append(start\_row)
except (SyntaxError, tokenize.TokenError):
pass
return line\_numbers
def shorten\_comment(line, max\_line\_length, last\_comment=False):
"""Return trimmed or split long comment line.
If there are no comments immediately following it, do a text wrap.
Doing this wrapping on all comments in general would lead to jagged
comment text.
"""
assert len(line) > max\_line\_length
line = line.rstrip()
# PEP 8 recommends 72 characters for comment text.
indentation = \_get\_indentation(line) + '# '
max\_line\_length = min(max\_line\_length,
len(indentation) + 72)
MIN\_CHARACTER\_REPEAT = 5
if (
len(line) - len(line.rstrip(line[-1])) >= MIN\_CHARACTER\_REPEAT and
not line[-1].isalnum()
):
# Trim comments that end with things like ---------
return line[:max\_line\_length] + '\n'
elif last\_comment and re.match(r'\s\*#+\s\*\w+', line):
split\_lines = textwrap.wrap(line.lstrip(' \t#'),
initial\_indent=indentation,
subsequent\_indent=indentation,
width=max\_line\_length,
break\_long\_words=False,
break\_on\_hyphens=False)
return '\n'.join(split\_lines) + '\n'
return line + '\n'
def normalize\_line\_endings(lines, newline):
"""Return fixed line endings.
All lines will be modified to use the most common line ending.
"""
return [line.rstrip('\n\r') + newline for line in lines]
def mutual\_startswith(a, b):
return b.startswith(a) or a.startswith(b)
def code\_match(code, select, ignore):
if ignore:
assert not isinstance(ignore, unicode)
for ignored\_code in [c.strip() for c in ignore]:
if mutual\_startswith(code.lower(), ignored\_code.lower()):
return False
if select:
assert not isinstance(select, unicode)
for selected\_code in [c.strip() for c in select]:
if mutual\_startswith(code.lower(), selected\_code.lower()):
return True
return False
return True
def fix\_code(source, options=None, encoding=None, apply\_config=False):
"""Return fixed source code.
"encoding" will be used to decode "source" if it is a byte string.
"""
options = \_get\_options(options, apply\_config)
if not isinstance(source, unicode):
source = source.decode(encoding or get\_encoding())
sio = io.StringIO(source)
return fix\_lines(sio.readlines(), options=options)
def \_get\_options(raw\_options, apply\_config):
"""Return parsed options."""
if not raw\_options:
return parse\_args([''], apply\_config=apply\_config)
if isinstance(raw\_options, dict):
options = parse\_args([''], apply\_config=apply\_config)
for name, value in raw\_options.items():
if not hasattr(options, name):
raise ValueError("No such option '{}'".format(name))
# Check for very basic type errors.
expected\_type = type(getattr(options, name))
if not isinstance(expected\_type, (str, unicode)):
if isinstance(value, (str, unicode)):
raise ValueError(
"Option '{}' should not be a string".format(name))
setattr(options, name, value)
else:
options = raw\_options
return options
def fix\_lines(source\_lines, options, filename=''):
"""Return fixed source code."""
# Transform everything to line feed. Then change them back to original
# before returning fixed source code.
original\_newline = find\_newline(source\_lines)
tmp\_source = ''.join(normalize\_line\_endings(source\_lines, '\n'))
# Keep a history to break out of cycles.
previous\_hashes = set()
if options.line\_range:
# Disable "apply\_local\_fixes()" for now due to issue #175.
fixed\_source = tmp\_source
else:
pep8\_options = {
'ignore': options.ignore,
'select': options.select,
'max\_line\_length': options.max\_line\_length,
'hang\_closing': options.hang\_closing,
}
sio = io.StringIO(tmp\_source)
contents = sio.readlines()
results = \_execute\_pep8(pep8\_options, contents)
codes = {result['id'] for result in results
if result['id'] in SELECTED\_GLOBAL\_FIXED\_METHOD\_CODES}
# Apply global fixes only once (for efficiency).
fixed\_source = apply\_global\_fixes(tmp\_source,
options,
filename=filename,
codes=codes)
passes = 0
long\_line\_ignore\_cache = set()
while hash(fixed\_source) not in previous\_hashes:
if options.pep8\_passes >= 0 and passes > options.pep8\_passes:
break
passes += 1
previous\_hashes.add(hash(fixed\_source))
tmp\_source = copy.copy(fixed\_source)
fix = FixPEP8(
filename,
options,
contents=tmp\_source,
long\_line\_ignore\_cache=long\_line\_ignore\_cache)
fixed\_source = fix.fix()
sio = io.StringIO(fixed\_source)
return ''.join(normalize\_line\_endings(sio.readlines(), original\_newline))
def fix\_file(filename, options=None, output=None, apply\_config=False):
if not options:
options = parse\_args([filename], apply\_config=apply\_config)
original\_source = readlines\_from\_file(filename)
fixed\_source = original\_source
if options.in\_place or output:
encoding = detect\_encoding(filename)
if output:
output = LineEndingWrapper(wrap\_output(output, encoding=encoding))
fixed\_source = fix\_lines(fixed\_source, options, filename=filename)
if options.diff:
new = io.StringIO(fixed\_source)
new = new.readlines()
diff = get\_diff\_text(original\_source, new, filename)
if output:
output.write(diff)
output.flush()
return diff
elif options.in\_place:
fp = open\_with\_encoding(filename, encoding=encoding, mode='w')
fp.write(fixed\_source)
fp.close()
original = "".join(original\_source).splitlines()
fixed = fixed\_source.splitlines()
if original != fixed:
return fixed\_source
else:
return ''
else:
if output:
output.write(fixed\_source)
output.flush()
return fixed\_source
def global\_fixes():
"""Yield multiple (code, function) tuples."""
for function in list(globals().values()):
if inspect.isfunction(function):
arguments = \_get\_parameters(function)
if arguments[:1] != ['source']:
continue
code = extract\_code\_from\_function(function)
if code:
yield (code, function)
def \_get\_parameters(function):
# pylint: disable=deprecated-method
if sys.version\_info.major >= 3:
# We need to match "getargspec()", which includes "self" as the first
# value for methods.
# https://bugs.python.org/issue17481#msg209469
if inspect.ismethod(function):
function = function.\_\_func\_\_
return list(inspect.signature(function).parameters)
else:
return inspect.getargspec(function)[0]
def apply\_global\_fixes(source, options, where='global', filename='',
codes=None):
"""Run global fixes on source code.
These are fixes that only need be done once (unlike those in
FixPEP8, which are dependent on pycodestyle).
"""
if codes is None:
codes = []
if any(code\_match(code, select=options.select, ignore=options.ignore)
for code in ['E101', 'E111']):
source = reindent(source,
indent\_size=options.indent\_size)
for (code, function) in global\_fixes():
if code.upper() in SELECTED\_GLOBAL\_FIXED\_METHOD\_CODES \
and code.upper() not in codes:
continue
if code\_match(code, select=options.select, ignore=options.ignore):
if options.verbose:
print('---> Applying {} fix for {}'.format(where,
code.upper()),
file=sys.stderr)
source = function(source,
aggressive=options.aggressive)
source = fix\_2to3(source,
aggressive=options.aggressive,
select=options.select,
ignore=options.ignore,
filename=filename,
where=where,
verbose=options.verbose)
return source
def extract\_code\_from\_function(function):
"""Return code handled by function."""
if not function.\_\_name\_\_.startswith('fix\_'):
return None
code = re.sub('^fix\_', '', function.\_\_name\_\_)
if not code:
return None
try:
int(code[1:])
except ValueError:
return None
return code
def \_get\_package\_version():
packages = ["pycodestyle: {}".format(pycodestyle.\_\_version\_\_)]
return ", ".join(packages)
def create\_parser():
"""Return command-line parser."""
parser = argparse.ArgumentParser(description=docstring\_summary(\_\_doc\_\_),
prog='autopep8')
parser.add\_argument('--version', action='version',
version='%(prog)s {} ({})'.format(
\_\_version\_\_, \_get\_package\_version()))
parser.add\_argument('-v', '--verbose', action='count',
default=0,
help='print verbose messages; '
'multiple -v result in more verbose messages')
parser.add\_argument('-d', '--diff', action='store\_true',
help='print the diff for the fixed source')
parser.add\_argument('-i', '--in-place', action='store\_true',
help='make changes to files in place')
parser.add\_argument('--global-config', metavar='filename',
default=DEFAULT\_CONFIG,
help='path to a global pep8 config file; if this file '
'does not exist then this is ignored '
'(default: {})'.format(DEFAULT\_CONFIG))
parser.add\_argument('--ignore-local-config', action='store\_true',
help="don't look for and apply local config files; "
'if not passed, defaults are updated with any '
"config files in the project's root directory")
parser.add\_argument('-r', '--recursive', action='store\_true',
help='run recursively over directories; '
'must be used with --in-place or --diff')
parser.add\_argument('-j', '--jobs', type=int, metavar='n', default=1,
help='number of parallel jobs; '
'match CPU count if value is less than 1')
parser.add\_argument('-p', '--pep8-passes', metavar='n',
default=-1, type=int,
help='maximum number of additional pep8 passes '
'(default: infinite)')
parser.add\_argument('-a', '--aggressive', action='count', default=0,
help='enable non-whitespace changes; '
'multiple -a result in more aggressive changes')
parser.add\_argument('--experimental', action='store\_true',
help='enable experimental fixes')
parser.add\_argument('--exclude', metavar='globs',
help='exclude file/directory names that match these '
'comma-separated globs')
parser.add\_argument('--list-fixes', action='store\_true',
help='list codes for fixes; '
'used by --ignore and --select')
parser.add\_argument('--ignore', metavar='errors', default='',
help='do not fix these errors/warnings '
'(default: {})'.format(DEFAULT\_IGNORE))
parser.add\_argument('--select', metavar='errors', default='',
help='fix only these errors/warnings (e.g. E4,W)')
parser.add\_argument('--max-line-length', metavar='n', default=79, type=int,
help='set maximum allowed line length '
'(default: %(default)s)')
parser.add\_argument('--line-range', '--range', metavar='line',
default=None, type=int, nargs=2,
help='only fix errors found within this inclusive '
'range of line numbers (e.g. 1 99); '
'line numbers are indexed at 1')
parser.add\_argument('--indent-size', default=DEFAULT\_INDENT\_SIZE,
type=int, help=argparse.SUPPRESS)
parser.add\_argument('--hang-closing', action='store\_true',
help='hang-closing option passed to pycodestyle')
parser.add\_argument('--exit-code', action='store\_true',
help='change to behavior of exit code.'
' default behavior of return value, 0 is no '
'differences, 1 is error exit. return 2 when'
' add this option. 2 is exists differences.')
parser.add\_argument('files', nargs='\*',
help="files to format or '-' for standard in")
return parser
def parse\_args(arguments, apply\_config=False):
"""Parse command-line options."""
parser = create\_parser()
args = parser.parse\_args(arguments)
if not args.files and not args.list\_fixes:
parser.error('incorrect number of arguments')
args.files = [decode\_filename(name) for name in args.files]
if apply\_config:
parser = read\_config(args, parser)
args = parser.parse\_args(arguments)
args.files = [decode\_filename(name) for name in args.files]
if '-' in args.files:
if len(args.files) > 1:
parser.error('cannot mix stdin and regular files')
if args.diff:
parser.error('--diff cannot be used with standard input')
if args.in\_place:
parser.error('--in-place cannot be used with standard input')
if args.recursive:
parser.error('--recursive cannot be used with standard input')
if len(args.files) > 1 and not (args.in\_place or args.diff):
parser.error('autopep8 only takes one filename as argument '
'unless the "--in-place" or "--diff" args are '
'used')
if args.recursive and not (args.in\_place or args.diff):
parser.error('--recursive must be used with --in-place or --diff')
if args.in\_place and args.diff:
parser.error('--in-place and --diff are mutually exclusive')
if args.max\_line\_length <= 0:
parser.error('--max-line-length must be greater than 0')
if args.select:
args.select = \_split\_comma\_separated(args.select)
if args.ignore:
args.ignore = \_split\_comma\_separated(args.ignore)
elif not args.select:
if args.aggressive:
# Enable everything by default if aggressive.
args.select = {'E', 'W1', 'W2', 'W3', 'W6'}
else:
args.ignore = \_split\_comma\_separated(DEFAULT\_IGNORE)
if args.exclude:
args.exclude = \_split\_comma\_separated(args.exclude)
else:
args.exclude = {}
if args.jobs < 1:
# Do not import multiprocessing globally in case it is not supported
# on the platform.
import multiprocessing
args.jobs = multiprocessing.cpu\_count()
if args.jobs > 1 and not args.in\_place:
parser.error('parallel jobs requires --in-place')
if args.line\_range:
if args.line\_range[0] <= 0:
parser.error('--range must be positive numbers')
if args.line\_range[0] > args.line\_range[1]:
parser.error('First value of --range should be less than or equal '
'to the second')
return args
def read\_config(args, parser):
"""Read both user configuration and local configuration."""
try:
from configparser import ConfigParser as SafeConfigParser
from configparser import Error
except ImportError:
from ConfigParser import SafeConfigParser
from ConfigParser import Error
config = SafeConfigParser()
try:
config.read(args.global\_config)
if not args.ignore\_local\_config:
parent = tail = args.files and os.path.abspath(
os.path.commonprefix(args.files))
while tail:
if config.read([os.path.join(parent, fn)
for fn in PROJECT\_CONFIG]):
break
(parent, tail) = os.path.split(parent)
defaults = {}
option\_list = {o.dest: o.type or type(o.default)
for o in parser.\_actions}
for section in ['pep8', 'pycodestyle', 'flake8']:
if not config.has\_section(section):
continue
for (k, \_) in config.items(section):
norm\_opt = k.lstrip('-').replace('-', '\_')
if not option\_list.get(norm\_opt):
continue
opt\_type = option\_list[norm\_opt]
if opt\_type is int:
value = config.getint(section, k)
elif opt\_type is bool:
value = config.getboolean(section, k)
else:
value = config.get(section, k)
if args.verbose:
print("enable config: section={}, key={}, value={}".format(
section, k, value))
defaults[norm\_opt] = value
parser.set\_defaults(\*\*defaults)
except Error:
# Ignore for now.
pass
return parser
def \_split\_comma\_separated(string):
"""Return a set of strings."""
return {text.strip() for text in string.split(',') if text.strip()}
def decode\_filename(filename):
"""Return Unicode filename."""
if isinstance(filename, unicode):
return filename
return filename.decode(sys.getfilesystemencoding())
def supported\_fixes():
"""Yield pep8 error codes that autopep8 fixes.
Each item we yield is a tuple of the code followed by its
description.
"""
yield ('E101', docstring\_summary(reindent.\_\_doc\_\_))
instance = FixPEP8(filename=None, options=None, contents='')
for attribute in dir(instance):
code = re.match('fix\_([ew][0-9][0-9][0-9])', attribute)
if code:
yield (
code.group(1).upper(),
re.sub(r'\s+', ' ',
docstring\_summary(getattr(instance, attribute).\_\_doc\_\_))
)
for (code, function) in sorted(global\_fixes()):
yield (code.upper() + (4 - len(code)) \* ' ',
re.sub(r'\s+', ' ', docstring\_summary(function.\_\_doc\_\_)))
for code in sorted(CODE\_TO\_2TO3):
yield (code.upper() + (4 - len(code)) \* ' ',
re.sub(r'\s+', ' ', docstring\_summary(fix\_2to3.\_\_doc\_\_)))
def docstring\_summary(docstring):
"""Return summary of docstring."""
return docstring.split('\n')[0] if docstring else ''
def line\_shortening\_rank(candidate, indent\_word, max\_line\_length,
experimental=False):
"""Return rank of candidate.
This is for sorting candidates.
"""
if not candidate.strip():
return 0
rank = 0
lines = candidate.rstrip().split('\n')
offset = 0
if (
not lines[0].lstrip().startswith('#') and
lines[0].rstrip()[-1] not in '([{'
):
for (opening, closing) in ('()', '[]', '{}'):
# Don't penalize empty containers that aren't split up. Things like
# this "foo(\n )" aren't particularly good.
opening\_loc = lines[0].find(opening)
closing\_loc = lines[0].find(closing)
if opening\_loc >= 0:
if closing\_loc < 0 or closing\_loc != opening\_loc + 1:
offset = max(offset, 1 + opening\_loc)
current\_longest = max(offset + len(x.strip()) for x in lines)
rank += 4 \* max(0, current\_longest - max\_line\_length)
rank += len(lines)
# Too much variation in line length is ugly.
rank += 2 \* standard\_deviation(len(line) for line in lines)
bad\_staring\_symbol = {
'(': ')',
'[': ']',
'{': '}'}.get(lines[0][-1])
if len(lines) > 1:
if (
bad\_staring\_symbol and
lines[1].lstrip().startswith(bad\_staring\_symbol)
):
rank += 20
for lineno, current\_line in enumerate(lines):
current\_line = current\_line.strip()
if current\_line.startswith('#'):
continue
for bad\_start in ['.', '%', '+', '-', '/']:
if current\_line.startswith(bad\_start):
rank += 100
# Do not tolerate operators on their own line.
if current\_line == bad\_start:
rank += 1000
if (
current\_line.endswith(('.', '%', '+', '-', '/')) and
"': " in current\_line
):
rank += 1000
if current\_line.endswith(('(', '[', '{', '.')):
# Avoid lonely opening. They result in longer lines.
if len(current\_line) <= len(indent\_word):
rank += 100
# Avoid the ugliness of ", (\n".
if (
current\_line.endswith('(') and
current\_line[:-1].rstrip().endswith(',')
):
rank += 100
# Avoid the ugliness of "something[\n" and something[index][\n.
if (
current\_line.endswith('[') and
len(current\_line) > 1 and
(current\_line[-2].isalnum() or current\_line[-2] in ']')
):
rank += 300
# Also avoid the ugliness of "foo.\nbar"
if current\_line.endswith('.'):
rank += 100
if has\_arithmetic\_operator(current\_line):
rank += 100
# Avoid breaking at unary operators.
if re.match(r'.\*[(\[{]\s\*[\-\+~]$', current\_line.rstrip('\\ ')):
rank += 1000
if re.match(r'.\*lambda\s\*\\*$', current\_line.rstrip('\\ ')):
rank += 1000
if current\_line.endswith(('%', '(', '[', '{')):
rank -= 20
# Try to break list comprehensions at the "for".
if current\_line.startswith('for '):
rank -= 50
if current\_line.endswith('\\'):
# If a line ends in \-newline, it may be part of a
# multiline string. In that case, we would like to know
# how long that line is without the \-newline. If it's
# longer than the maximum, or has comments, then we assume
# that the \-newline is an okay candidate and only
# penalize it a bit.
total\_len = len(current\_line)
lineno += 1
while lineno < len(lines):
total\_len += len(lines[lineno])
if lines[lineno].lstrip().startswith('#'):
total\_len = max\_line\_length
break
if not lines[lineno].endswith('\\'):
break
lineno += 1
if total\_len < max\_line\_length:
rank += 10
else:
rank += 100 if experimental else 1
# Prefer breaking at commas rather than colon.
if ',' in current\_line and current\_line.endswith(':'):
rank += 10
# Avoid splitting dictionaries between key and value.
if current\_line.endswith(':'):
rank += 100
rank += 10 \* count\_unbalanced\_brackets(current\_line)
return max(0, rank)
def standard\_deviation(numbers):
"""Return standard deviation."""
numbers = list(numbers)
if not numbers:
return 0
mean = sum(numbers) / len(numbers)
return (sum((n - mean) \*\* 2 for n in numbers) /
len(numbers)) \*\* .5
def has\_arithmetic\_operator(line):
"""Return True if line contains any arithmetic operators."""
for operator in pycodestyle.ARITHMETIC\_OP:
if operator in line:
return True
return False
def count\_unbalanced\_brackets(line):
"""Return number of unmatched open/close brackets."""
count = 0
for opening, closing in ['()', '[]', '{}']:
count += abs(line.count(opening) - line.count(closing))
return count
def split\_at\_offsets(line, offsets):
"""Split line at offsets.
Return list of strings.
"""
result = []
previous\_offset = 0
current\_offset = 0
for current\_offset in sorted(offsets):
if current\_offset < len(line) and previous\_offset != current\_offset:
result.append(line[previous\_offset:current\_offset].strip())
previous\_offset = current\_offset
result.append(line[current\_offset:])
return result
class LineEndingWrapper(object):
r"""Replace line endings to work with sys.stdout.
It seems that sys.stdout expects only '\n' as the line ending, no matter
the platform. Otherwise, we get repeated line endings.
"""
def \_\_init\_\_(self, output):
self.\_\_output = output
def write(self, s):
self.\_\_output.write(s.replace('\r\n', '\n').replace('\r', '\n'))
def flush(self):
self.\_\_output.flush()
def match\_file(filename, exclude):
"""Return True if file is okay for modifying/recursing."""
base\_name = os.path.basename(filename)
if base\_name.startswith('.'):
return False
for pattern in exclude:
if fnmatch.fnmatch(base\_name, pattern):
return False
if fnmatch.fnmatch(filename, pattern):
return False
if not os.path.isdir(filename) and not is\_python\_file(filename):
return False
return True
def find\_files(filenames, recursive, exclude):
"""Yield filenames."""
while filenames:
name = filenames.pop(0)
if recursive and os.path.isdir(name):
for root, directories, children in os.walk(name):
filenames += [os.path.join(root, f) for f in children
if match\_file(os.path.join(root, f),
exclude)]
directories[:] = [d for d in directories
if match\_file(os.path.join(root, d),
exclude)]
else:
yield name
def \_fix\_file(parameters):
"""Helper function for optionally running fix\_file() in parallel."""
if parameters[1].verbose:
print('[file:{}]'.format(parameters[0]), file=sys.stderr)
try:
return fix\_file(\*parameters)
except IOError as error:
print(unicode(error), file=sys.stderr)
def fix\_multiple\_files(filenames, options, output=None):
"""Fix list of files.
Optionally fix files recursively.
"""
results = []
filenames = find\_files(filenames, options.recursive, options.exclude)
if options.jobs > 1:
import multiprocessing
pool = multiprocessing.Pool(options.jobs)
ret = pool.map(\_fix\_file, [(name, options) for name in filenames])
results.extend(filter(lambda x: x is not None, ret))
else:
for name in filenames:
ret = \_fix\_file((name, options, output))
if ret is None:
continue
if options.diff or options.in\_place:
if ret != '':
results.append(ret)
else:
original\_source = readlines\_from\_file(name)
if "".join(original\_source).splitlines() != ret.splitlines():
results.append(ret)
return results
def is\_python\_file(filename):
"""Return True if filename is Python file."""
if filename.endswith('.py'):
return True
try:
with open\_with\_encoding(
filename,
limit\_byte\_check=MAX\_PYTHON\_FILE\_DETECTION\_BYTES) as f:
text = f.read(MAX\_PYTHON\_FILE\_DETECTION\_BYTES)
if not text:
return False
first\_line = text.splitlines()[0]
except (IOError, IndexError):
return False
if not PYTHON\_SHEBANG\_REGEX.match(first\_line):
return False
return True
def is\_probably\_part\_of\_multiline(line):
"""Return True if line is likely part of a multiline string.
When multiline strings are involved, pep8 reports the error as being
at the start of the multiline string, which doesn't work for us.
"""
return (
'"""' in line or
"'''" in line or
line.rstrip().endswith('\\')
)
def wrap\_output(output, encoding):
"""Return output with specified encoding."""
return codecs.getwriter(encoding)(output.buffer
if hasattr(output, 'buffer')
else output)
def get\_encoding():
"""Return preferred encoding."""
return locale.getpreferredencoding() or sys.getdefaultencoding()
def main(argv=None, apply\_config=True):
"""Command-line entry."""
if argv is None:
argv = sys.argv
try:
# Exit on broken pipe.
signal.signal(signal.SIGPIPE, signal.SIG\_DFL)
except AttributeError: # pragma: no cover
# SIGPIPE is not available on Windows.
pass
try:
args = parse\_args(argv[1:], apply\_config=apply\_config)
if args.list\_fixes:
for code, description in sorted(supported\_fixes()):
print('{code} - {description}'.format(
code=code, description=description))
return EXIT\_CODE\_OK
if args.files == ['-']:
assert not args.in\_place
encoding = sys.stdin.encoding or get\_encoding()
# LineEndingWrapper is unnecessary here due to the symmetry between
# standard in and standard out.
wrap\_output(sys.stdout, encoding=encoding).write(
fix\_code(sys.stdin.read(), args, encoding=encoding))
else:
if args.in\_place or args.diff:
args.files = list(set(args.files))
else:
assert len(args.files) == 1
assert not args.recursive
ret = fix\_multiple\_files(args.files, args, sys.stdout)
if args.exit\_code and len(ret) > 0:
return EXIT\_CODE\_EXISTS\_DIFF
except KeyboardInterrupt:
return EXIT\_CODE\_ERROR # pragma: no cover
class CachedTokenizer(object):
"""A one-element cache around tokenize.generate\_tokens().
Original code written by Ned Batchelder, in coverage.py.
"""
def \_\_init\_\_(self):
self.last\_text = None
self.last\_tokens = None
def generate\_tokens(self, text):
"""A stand-in for tokenize.generate\_tokens()."""
if text != self.last\_text:
string\_io = io.StringIO(text)
self.last\_tokens = list(
tokenize.generate\_tokens(string\_io.readline)
)
self.last\_text = text
return self.last\_tokens
\_cached\_tokenizer = CachedTokenizer()
generate\_tokens = \_cached\_tokenizer.generate\_tokens
if \_\_name\_\_ == '\_\_main\_\_':
sys.exit(main())