利用 Python 进行生信分析 | 基础 + 实战

——Python for Bioinformatics

赵华男

2022 一起加油

自我介绍

赵华男

研究方向

生物信息学 | 基因编辑

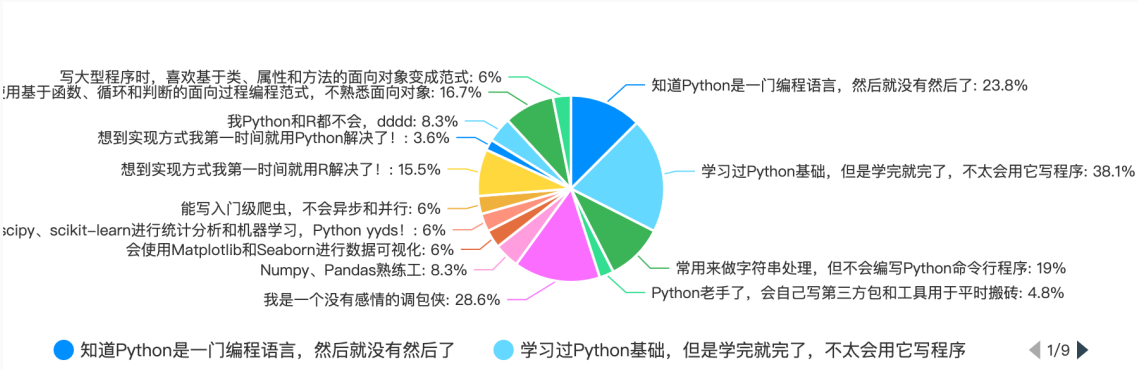
学习经历

- 2014 ~ 2019　西北农林科技大学 (学士)
- 2016 ~ 2018　中国农业大学 (交换生)
- 2019 ~ 至今　清华大学博士研究生 (PTN 项目)
- 2020 ~ 至今　北京大学 (联合培养)

第一节 课程介绍

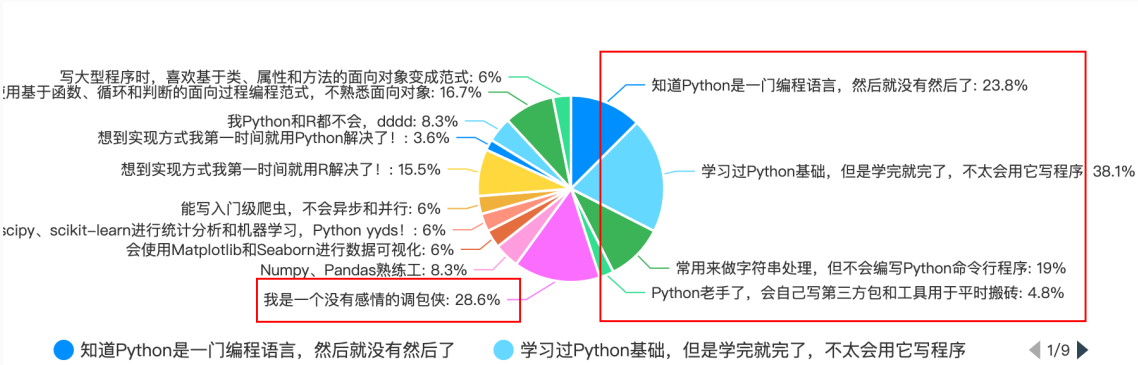
这门课的开设目的

教大家使用 Python 写程序



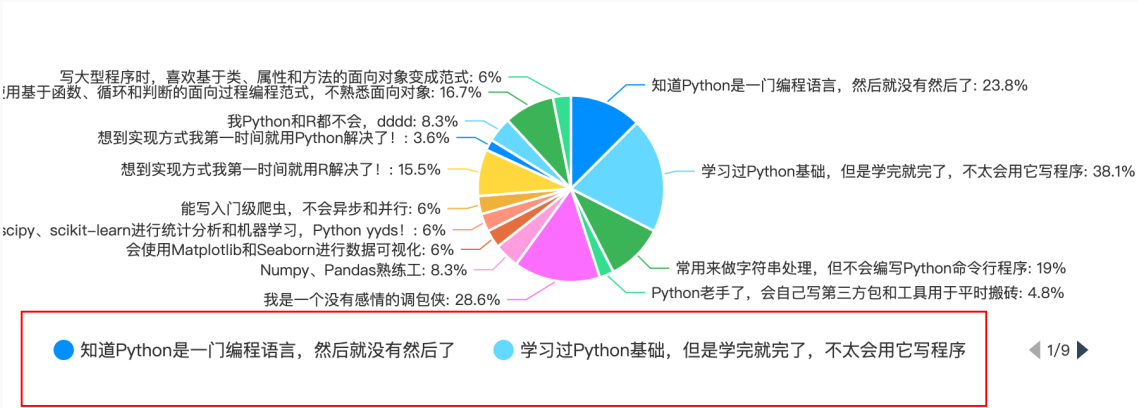
这门课的开设目的

教大家使用 Python 写程序



这门课的开设目的

教大家使用 Python 写程序 → 基础 + 有难度的几个实战项目



课程设计

- 操作系统和编程语言 (Shell, PATH)
- Python 基础知识 + Python 进阶知识
- 编程实战 (核心)
  - 序列文件的处理 (FASTA, FASTQ)
  - BED 文件的操作与处理 (BED)
  - 基因注释文件处理 (GFF, GTF)
  - BAM 文件的解析与操作 (SAM, BAM)
  - 支持多核计算程序的编写
  - 复杂命令行工具的编写与搭建

Thank you!

第二节　操作系统和编程语言

Python 语言的特点

- Guido van Rossum 在 1989 年创造了 Python, 在 1991 年将 Python 首次公开发行
- 跨平台、开源、解释型语言、高级语言
- 源代码可见
- 第三方包种类多, 可用底层语言 (如 C、Rust) 重写关键代码
- 开发效率高，执行效率低

Python 语言的特点

- Python 开发效率高，执行效率低 (科研, 数据分析)
- C 开发效率低, 执行效率高 (高频量化交易)

执行效率对比

- C 10000 行 0.01s
- Java 1000 行 0.05s
- Python 100 行 0.1s

Python 擅长的领域

- 爬虫
- Web 后端开发
- 自动化运维、自动化测试
- 数据科学
- 机器学习

Python 在生物信息学中的应用

- **数据获取 (爬取 UniProt)**
  - 爬虫
- **Web 前后端开发**
  - Django, Flask, 数据库...
- **工作流程**
  - Jupyterlab
  - Snakemake
- **字符串处理**
  - FASTA, FASTQ, BED-like, BAM/SAM
  - Biopython, Pysam
- **数据分析机器学习**
  - **数据清洗, 数据分析, 可视化**
  - 特征工程, 机器学习

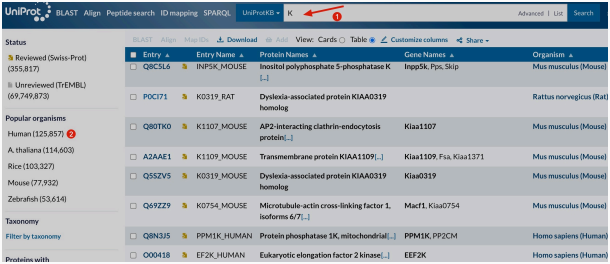


图 1: UniProt 蛋白数据库界面

Python 在生物信息学中的应用

- **数据获取**
  - 爬虫
- **Web 前后端开发**
  - Django, Flask, 数据库...
- **工作流程**
  - Jupyterlab
  - Snakemake
- **字符串处理**
  - FASTA, FASTQ, BED-like, BAM/SAM
  - Biopython, Pysam
- **数据分析机器学习 (AlphaFold2$_{TensorFlow}$)**
  - **数据清洗, 数据分析, 可视化**
  - 特征工程, 机器学习

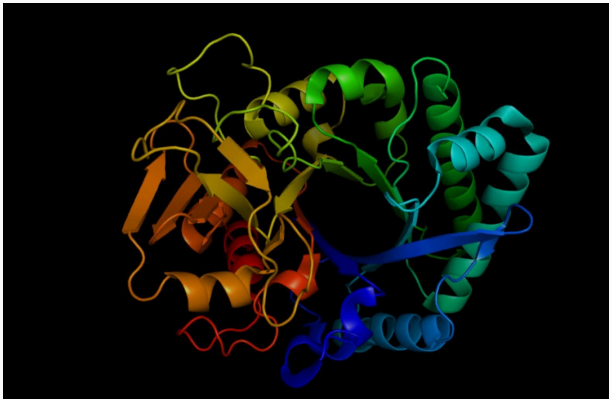


图 2: AlphaFold2 蛋白结构预测

Python 在生物信息学中的应用

- **数据获取**
  - 爬虫
- Web 前后端开发 (AnnoLnc2)
  - Django, Flask, 数据库...
- **工作流程**
  - Jupyterlab
  - Snakemake
- **字符串处理**
  - FASTA, FASTQ, BED-like, BAM/SAM
  - Biopython, Pysam
- **数据分析机器学习**
  - **数据清洗, 数据分析, 可视化**
  - 特征工程, 机器学习

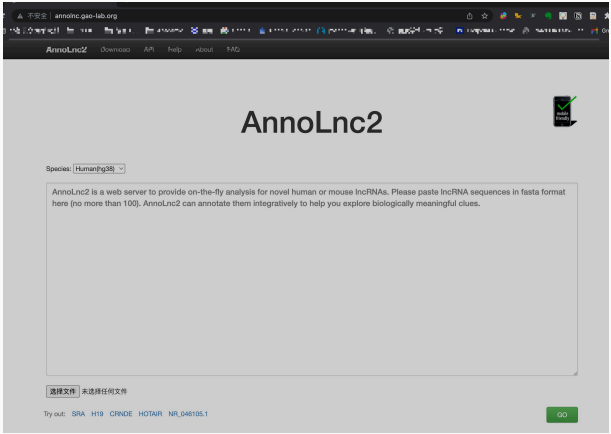


图 3: AnnoLnc2 lncRNA 网页端分析工具

Python 在生物信息学中的应用

- **数据获取**
  - 爬虫
- **Web 前后端开发**
  - Django, Flask, 数据库...
- **工作流程**
  - Jupyterlab
  - Snakemake
- **字符串处理 (强项)**
  - FASTA, FASTQ, BED-like, BAM/SAM
  - Biopython, Pysam
- **数据分析机器学习**
  - 数据清洗, 数据分析, 可视化
  - 特征工程, 机器学习

从 Windows 1.0 到 Windows 11

图 4: Windows 11

图 5: MS-DOS 命令行界面

现在在 Windows 系统中的 PowerShell 已经取代了 CMD 成为新一代壳程序 (Shell), 但是 CMD 仍然集成在操作系统中

https://zh.wikipedia.org/wiki/Microsoft_Windows
https://zh.wikipedia.org/wiki/File:MS-DOS_Deutsch.png
https://zh.wikipedia.org/wiki/Windows_11#/media/File:Windows_11_Desktop.png

Windows, Linux, MacOS

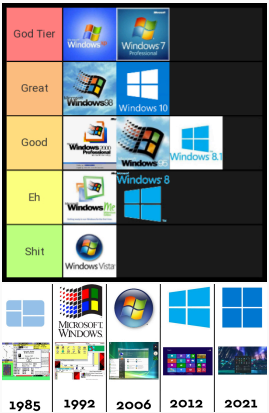


图 6: Windows

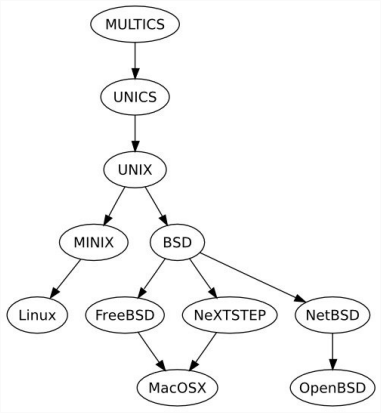


图 7: Linux, MacOS 起源于 UNIX

Windows, Linux, MacOS

Linux (Open source)

- 上百种不同的发行版...
- Ubuntu/Debian
- CentOS
- Android (Google Inc.)
- Shell: (sh, bash, zsh...)

MacOS (Apple Inc.)

- Monterey
- Shell: (sh, bash, zsh...)

图 8: Linux 发行版本

Windows 与 Unix-like 系统之间的区别

特点	Windows	Unix-like
参考版本	Windows10	Ubuntu20.04
是否开源	否	是
路径表示	反斜杠	斜杠
文件结构	分区管理	单一树状
Shell 命令	CMD 风格	Bash 风格

Windows:

C:\User\zhaohuanan\Document\a.txt

D:\somewhere\Document\b.txt

Ubuntu:

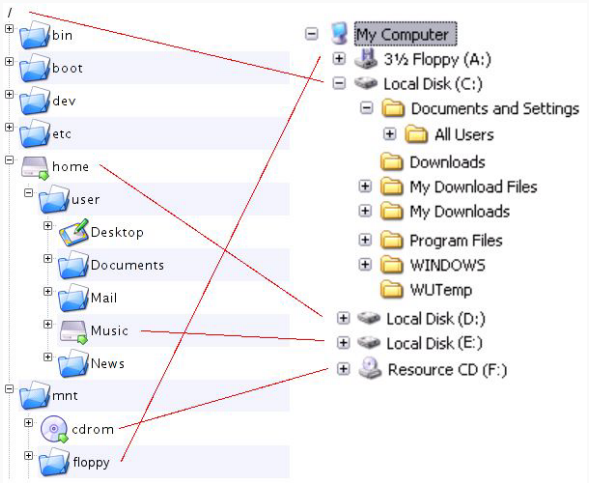
/home/zhaohuanan/Document/a.txt

/home/someone/Document/b.txt

Shell 命令举例

操作	Windows CMD 命令	Ubuntu Bash 命令	相似	不同
切换目录	cd	cd	√	
打印当前目录	cd	pwd		√
清除屏幕	cls	clear		√
列出文件与子目录	dir	ls ls -l		√
列出所有文件与子目录	dir /a	ls -a ls -la		√
创建目录	mkdir md	mkdir mkdir -p	√	
删除目录	rmdir rd	rmdir rm -rf	√	
新建文件	cd .> | notepad	touch		√
删除文件	del	rm		√
拷贝文件	copy	cp		√
在 Shell 中打印文件	type	cat		√
移动文件	move	mv		√
展示目录结构	tree	tree	√	

文件系统结构


/
bin
boot
dev
etc
home
user
Desktop
Documents
Mail
Music
News
mnt
cdrom
floppy
My Computer
3½ Floppy (A:)
Local Disk (C:)
Documents and Settings
All Users
Downloads
My Download Files
My Downloads
Program Files
WINDOWS
WUTemp
Local Disk (D:)
Local Disk (E:)
Resource CD (F:)

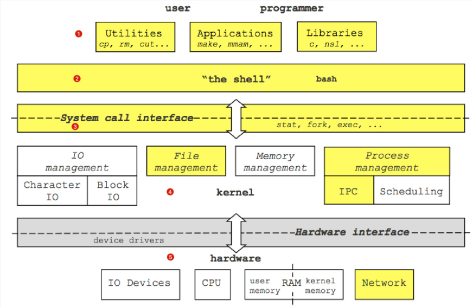
路径演示

- 演示 Windows, MacOS, Linux(以 Ubuntu 为例) 进入文件路径
- 打印当前目录, 切换目录
- 列出文件与子目录, 列出所有文件与子目录, 创建目录, 删除目录
- 新建文件, 拷贝文件, 移动文件, 删除文件
- 在 Shell 中打印文件, 清除屏幕
- 展示目录结构

演示结论

- 类 Unix 系统和 Windows 系统在具体的特点上区别很大
  - 路径分隔符不同
  - 文件系统的结构不同
  - Shell 命令不同
- 但是这些操作系统的抽象层面上的逻辑都是类似的!
  - Shell 命令不同, 但具有相同或相似的功能, 如 type 和 cat
  - 文件系统中的操作具有共性
    - 文件/文件夹的创建/拷贝/移动/删除
    - 目录切换
    - ...

Unix 系统层级



1. 用户使用 Shell 命令，利用 Shell 程序来对操作系统发出指令
2. Shell 程序是用户与操作系统交互的媒介
3. 操作系统负责调度文件，管理内存，管理进程等
4. 内核负责将系统层面的指令传递给硬件，如 CPU，内存条，网卡，硬盘等
5. 硬件，最终完成计算，读取和存储，网络传输等任务的元件

- **用户空间**
  - 定义用户可访问的应用程序、库和标准实用程序
  - Shell 也在 “用户空间” 中运行
- **内核空间**
  - 管理用户操作和硬件之间接口的操作系统的操作
  - 操作系统的核心部分，其主要工作是将用户应用程序与底层硬件配对，并允许多个程序共享单个硬件组件
- **硬件**
  - 计算机的底层物理组件
  - 输入/ 输出设备，如键盘和显示器、进行计算的 CPU、内存组件和网络接口

https://classes.adamaviv.com/ic221/s18/units/01/unit.html

Linux 的应用场景

- **桌面应用**
  - Ubuntu
  - Archlinux
- **嵌入式应用**
  - 电梯、汽车、玩具、灯具、冰箱、空调、电饭煲等，都属于嵌入式概念的应用
- **服务器应用**
  - 开源, 免费, 无需考虑商业软件授权问题
  - 高稳定性, 高可靠性
  - 硬件要求低

“the Shell”

概念: 在计算机科学中, Shell 俗称壳 (用来区别于核), 是指 “为使用者提供操作界面” 的软件。

- **图形界面 Shell**
  - Graphical User Interface Shell, GUI Shell
  - Windows
    - Windows Explorer
  - Linux
    - X-Window
    - GENOME
    - ...
- **命令行 Shell**
  - Command Line Interface Shell, CLI Shell
  - Windows
    - CMD
    - PowerShell
  - Linux
    - sh
    - bash
    - zsh
    - ...

传统意义上的 Shell 指的是命令行式的 Shell

Shell 是一个操作界面, 里面的每一个命令是怎么来的?

命令与 PATH

PATH 是一个环境变量, PATH 中包含了可执行文件 的搜索路径

当要求系统运行一个程序而没有告诉它程序所在的完整路径时, 系统除了在当前目录下面寻找此程序外, 还应到 “PATH” 中指定的路径去找。用户通过设置环境变量, 来更好的运行进程。

```
(base) ~ echo $PATH
/usr/local/Caskroom/miniconda/base/bin:/usr/local/Caskroom/
miniconda/base/condabin:/Users/zhaohuanan/NutstoreFiles/MyN
utstore/Coding/bin:/usr/local/bin:/usr/local/sbin:/usr/loca
l/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin:/us
r/local/share/dotnet:~/.dotnet/tools:/Library/Apple/usr/bin
:/Users/zhaohuanan/.cargo/bin:/Users/zhaohuanan/.emacs.d/bi
n
(base) ~
```

PATH 的演示

- MacOS/Linux
  - $PATH
- Windows
  - %PATH%

演示结论

- 我们使用的命令, 其本质是可执行文件
  - 可执行文件具有完整的路径
  - 可以直接使用命令的名称, 是因为其所在文件目录在 PATH 中
- 通过 PATH 中添加路径, 可以省略可执行文件的完整路径
  - 当系统找不到用户输入的可执行文件时, 会在 PATH 中从前到后查找
  - 当在 PATH 中所有的路径下都找不到用户的可执行文件, 会报错没有此可执行文件
- 安装程序时, 可以将可执行文件目录加入 PATH, 以方便使用此程序

编译型语言

编译型语言要求使用编译器一次性将所有源代码编译为一个可执行程序，一次编译可重复执行。

- **编译型语言一般不能跨平台**
  - 编译出来的可执行程序不能跨平台：因为不同操作系统对可执行文件有着不同的要求，彼此之间不能兼容。
  - 源代码不能跨平台: 不同操作系统下的函数、变量、api 等可能会有不同。
- **代表语言：**
  - C、C++
  - Golang
  - Rust
  - Java?

解释型语言

解释型语言是使用解释器一边执行一边转换，用到些源代码就转换哪些，不会生成可执行程序。

- **解释型语言一般可以跨平台**
  - 跨平台是指源代码可以跨平台，解释器是不能跨平台的
  - 源代码在不同操作系统中运行的结果相同
  - 一个语言可以有不同的解释器用不同的语言实现这个解释器
- **代表语言：**
  - C#
  - JavaScript
  - PHP
  - Ruby
  - R
  - Python

编译器与解释器

- **编译器 (编译型语言)**
  - 编译器在编译的过程中，读入源程序文件，输出一份等价的二进制可执行文件，就和笔译工作者一样，他们都会输出一份翻译后的文件。
- **解释器 (解释型语言)**
  - 解释器在解释的过程中，读入源程序文件，输出的是执行的结果，就和口译工作者一样，他们输出的是已经完成翻译的结果。

输出的不同是这两者最大的区别，一个会输出用于执行的文件，另一个只会输出运行的结果。

https://zhuanlan.zhihu.com/p/389371438

Python 是一种解释型语言 (了解)

超纲内容, 入门同学可以先听不懂!

- 导入大型包的时候第一次导入很慢, 以后会很快的原因?
- Python 并非完全是解释性语言, 它是有编译的:
  - Python 在解释源程序时是分成两个步骤的:
    - 首先处理 py 中的源代码, 编译生成一个二进制 pyc字节码文件 (main + module)
    - 再对字节码进行处理, 才会生成 CPU 能够识别的机器码
    - 有了 module 的字节码文件之后, 下一次运行程序时, 如果在上次保存字节码之后没有修改过源代码, Python 将会加载.pyc 文件并跳过编译 module 的字节码这个步骤
    - 当 Python 重编译时, 它会自动检查 module 源文件和字节码文件的时间戳
    - 如果你又修改了 module 源代码, 下次程序运行时,module 的字节码 pyc 文件将自动重新创建
  - 相对于 py 文件来说, 编译成 pyc 本质上和 py 没有太大区别, 只是对于这个模块的加载速度提高了; 并没有提高代码的执行速度
  - 通常情况下不用主动去编译 pyc 文件, 除非需要隐藏源代码保持私密性
- ".pyc" 文件可以对 module 导入进行加速, Python CLI 程序设计原则:
  - 在 import 别的 py 文件 (module, 模块) 时, 那个 py 文件会被存一份 pyc 加速下次装载
  - 而主文件因为是直接执行而不是 import 导入所以不会保留 pyc
  - 这也是为什么写大程序原则是 CLI 入口代码尽量少的原因:
    - 将主要代码都构建在模块中以加速程序启动
    - 因为 CLI 入口是当前运行的主程序, 并不会生成 pyc!
- 演示

接下来的要求

- 听课要求 (零基础)
  - 手机 No!
  - iPad No!
  - 电脑 Yes!
- 养成记笔记的习惯
  - 使用 Markdown 来记笔记
  - 使用一个开箱即用的笔记工具
- 心态要求
  - 看别人码代码是很枯燥的
  - 动手自己跟着码代码, 获得成就感
  - 程序员/生信工作者 90% 的时间是在解决问题的路上以及学习, 只有 10% 的时间在流畅地写代码, 要从 Debug 的过程中得到进步和获得快乐!
- 不懂的概念和知识点
  - 读报错信息, 动脑思考, 如果还不行 ↓
  - 百度, 如果还不行 ↓
  - Google, 如果还不行 ↓
  - 群里问, 请教师兄师姐

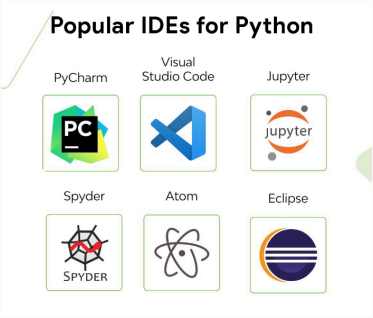
集成开发环境-IDE

集成开发环境 (Integrated Development Environment, IDE):

用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面等工具。集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套件。所有具备这一特性的软件都可以叫集成开发环境。

流行的 Python IDE:

- PyCharm (√)
- VS Code
- Jupyter Lab (√)
- Spyder
- IDLE, Sublime, Vim, Emacs...



https://gknxt.com/python/whatisanide.php

Python 的 “解释器”

- Python 解释器 (Interpreter)
  - 解释执行 Python 源代码
  - 官方版本的 Python 解释器是由 C 语言开发的 (CPython)
  - 其他版本的 Python 解释器:
    - IPython
    - PyPy
    - Jython
    - IronPython
- 当我们说, “安装 Python”, 意思是安装 Python 的解释器
  - 默认情况我们认为是安装官方版本的 CPython 解释器
  - 默认我们使用的也是 CPython
  - IPython 以及 Jupyter Notebook/Lab 是基于 IPython 的
- IDE
  - PyCharm、VS Code、Spyder、Jupyter-lab
    - 编写 Python 代码的工具
    - 整合了 Python 解释器
    - 代码提示功能
    - 减轻工作量

安装官方的 Python 解释器

https://www.python.org/

- Windows
- MacOS
- Linux

安装官方的 Python 解释器–Windows

安装官方的 Python 解释器–Linux

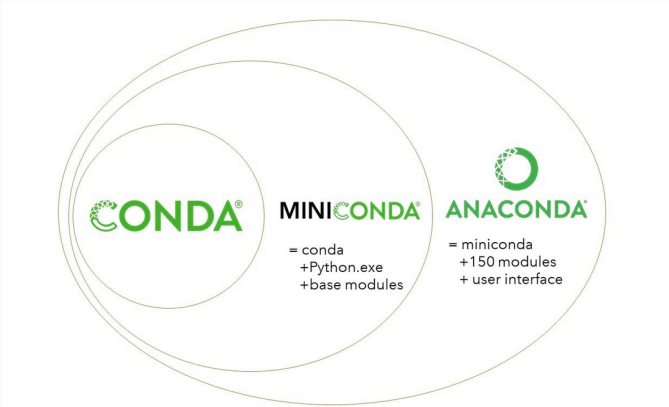
安装官方的 Python 解释器–MacOS

Python 的包管理工具

- pip
  - 查询包
  - 安装包
  - 卸载包
  - 一定程度的自动配置环境依赖功能
- venv
  - 创建 Python 的虚拟环境
  - 其余功能类似 pip
- conda
  - 查询、安装、卸载 Python 包
  - 创建、切换、管理 Python 运行环境
  - 命令行工具安装（生信、数据科学必会工具）
  - 强大的自动配置环境依赖功能

Conda 简介

- Conda 是一款环境管理工具
  - 最流行的 Python 环境管理工具之一
  - 开源的软件包管理系统和环境管理系统,
    - 用于安装多个版本的软件包及其依赖关系, 并在不同环境间切换
    - Conda 是为 Python 程序创建的, 也可以打包和分发其他软件
  - Linux, MacOS 和 Windows 跨平台
- Conda
  - Anaconda
  - Miniconda



安装和配置 conda 环境

下载: tuna → conda → 清华大学 tuna 镜像站 → 搜索 conda → 选择 anaconda → 选择 miniconda → 选择 “latest” 版本!

- Windows
- MacOS
- Linux
- **用法**
  - conda init 初始化 Shell 环境
  - 创建一个指定 Python 版本的 Conda 环境
  - 切换不同 Python 版本的 Conda 环境
  - 对不同 Conda 环境安装不同版本的 Python 包
  - 对一个 Conda 环境安装其他软件

https://conda.io/projects/conda/en/latest/user-guide/install/index.html
https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/
https://mirrors.tuna.tsinghua.edu.cn/anaconda/miniconda/

安装和配置 conda 环境–Windows

安装和配置 conda 环境–MacOS

安装和配置 conda 环境–Linux

Conda 源–以 MacOS 为例

```
# 清除之前残留的conda channels
rm -rf ~/.condarc
# 按顺序依次添加channel, 尽可能使用官方源
# 其他源常常在同步库的时候发生
# md5 值校验错误装不上包的问题!!!!
conda config --add channels defaults
conda config --add channels conda-forge
conda config --add channels bioconda
conda config --add channels r
# 备用, 清华源
# https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/
# https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/msys2/
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/
#设置搜索是显示通道地址
conda config --set show_channel_urls yes
#查看当前conda配置
conda config --show channels
conda config --get channels
```

```
cat ~/.condarc
# return
channels:
    - r
    - bioconda
    - conda-forge
    - microsoft
    - defaults
```

pip 源

```
# PyPI 镜像在每次同步成功后间隔 5 分钟同步一次。
# 临时使用，注意，simple 不能少，是 https 而不是 http
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple some-package

# 设为默认
# 升级 pip 到最新的版本 (>=10.0.0) 后进行配置：
python -m pip install --upgrade pip
pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
# 如果您到 pip 默认源的网络连接较差，临时使用本镜像站来升级 pip:
python -m pip install -i https://pypi.tuna.tsinghua.edu.cn/simple --upgrade pip
```

https://mirrors.tuna.tsinghua.edu.cn/anaconda/

Conda, Pip 的扩展用例

```
# install mamba
conda install -c conda-forge mamba
mamba update -n base -c defaults conda
mamba install pandas
# conda 导出环境/导入环境
## 导出当前环境：
conda env export > requirements_conda.yml
## 导入环境：
conda env create -f requirements_conda.yml
## 备注：一些找不到的小组件直接删掉后再安装
# 滚回某个环境
conda list --revisions
conda install --revision=n
# 删除某环境
conda env remove -n learn

# pip 导出环境/导入环境
## 导出当前环境：
pip freeze > requirements.txt
## 导入环境：
pip install -r requirements.txt
```

安装和配置 JupyterLab 环境

- 使用 Conda 安装和配置 JupyterLab
  -
- JupyterLab 基本用法
  - ipynb
  - terminal

install JupyterLab–Windows

install JupyterLab

```
conda install jupyterlab nodejs # nodejs > 12.14
pip install jupyterlab_theme_hale # theme
jupyter-lab --no-browser --port 8888 # run
# 启用extension manager

# func: Settings Editor Form UI
# setting:
{
    "settingEditorType": "json"
}
# func: File Browser
# setting:
{
    "navigateToCurrentDirectory": true,
    "useFuzzyFilter": true,
    "showLastModifiedColumn": true,
    "showHiddenFiles": true,
}
```

```
# func: Text Editor
# setting:
{
    "editorConfig": {
        "autoClosingBrackets": true, // 补全括号
        "codeFolding": true, // 代码折叠
        "cursorBlinkRate": 530,
        "fontFamily": null,
        "fontSize": null,
        "insertSpaces": true,
        "lineHeight": null,
        "lineNumbers": true,
        "lineWrap": "on",
        "matchBrackets": true,
        "readOnly": false,
        "tabSize": 4,
        "rulers": [],
        "showTrailingSpace": false,
        "wordWrapColumn": 80
    }
}
```

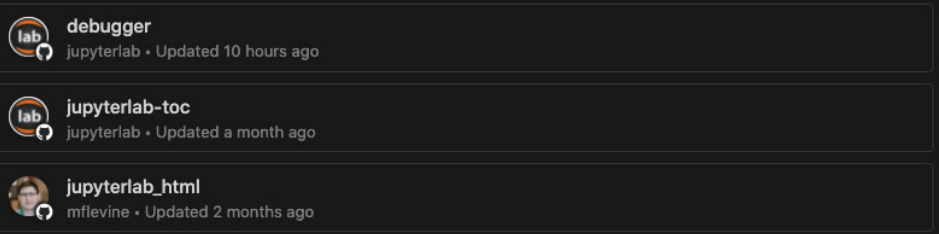
install JupyterLab–MacOS

已经整合的优秀插件

已经吸纳入Jupyterlab本体的优秀插件

（自带，不用安装）

老版本的 jupyter-lab v2.*可能需要安装



自动补全插件

```
pip install jupyterlab-lsp
pip install python-lsp-server pyright
conda install r-languageserver # 如果需要用 R


# func: Code Syntax
# 启用或者关闭 lsp 自动补全
# true 为关闭，false 为开启
# setting:
{
    "disable": false,
}
```

```
# func: Language Server
# setting:
{
    "language_servers": {
    "pyls": {
        "serverSettings": {
        "pyls.plugins.pydocstyle.enabled": true,
        "pyls.plugins.pyflakes.enabled": false,
        "pyls.plugins.flake8.enabled": true
        }
    },
    # 如果需要用 R
    "r-languageserver": {
        "serverSettings": {
        "r.lsp.debug": false,
        "r.lsp.diagnostics": false
        }
    }
    }
}
```

Q & A!

Thank you!

第三节 Python 基础知识

Hello World

- 提示: 输入法使用英文!
- 两种运行方式
  - python 的 shell 页面
    - “交互式”, 输入一行, 按回车, 返回一次结果, exit() 退出 (演示)
    - 应用” 在 python 中测试少量代码
  - python 源代码文件 “.py”
    - 编辑好每一行代码, 在命令行全部运行 (演示)
    - 本质: 使用安装在指定路径下的 python 解释器 运行指定目录下的 “.py” 源代码文件 (文本文件)(演示)
  - 源代码文件本质是 “文本文件”(可以使用 cat 或者 type 命令打印到命令行)
- 知识点
  - “123” 字符串
  - 123 整数
  - # 注释
    - 注释一行
    - 行内注释
- 使用不同 IDE 进行演示并说明不同 IDE 运行逻辑

开讲之前!

- Python 基础
  - Jupyterlab 学习环境
  - 实战课, 基础部分知识点 (全面覆盖) + 练习
  - 在直播课程中缓冲的时间较少 (每个练习和习题的用意)
  - 在录播课程中反复观看和揣摩
- Python 进阶
  - 安装 PyCharm
  - 适应 PyCharm 开发环境, 代码规范
  - 逐步向本课程实战部分过度

变量与数据类型

变量: 变量是存放数据值的容器

- 与其他编程语言不同, Python 没有声明变量的命令
- 首次为其赋值时，才会创建变量
- 变量不需要使用任何特定类型声明，甚至可以在设置后更改其类型
- 字符串变量可以使用单引号, 双引号, 三引号进行声明

Python 变量命名规则

- 只能包含字母数字字符和下划线 (A-z、0-9 和 _)
- 必须以字母或下划线字符开头, 不能以数字开头
- 变量名称区分大小写 (age、Age 和 AGE 是三个不同的变量)
- 如果使用关键字作为变量名?
- Jupyterlab 演示

Python 变量赋值规则

- 常规赋值
- 向多个变量赋值（相同值）
- 向多个变量赋值（不同值），解包（了解）
- 问题：如何将两个变量的值互换?

Python 变量的打印

- 打印一个变量
- 打印多个变量
- 将变量连接到字符串后，进行打印
  - 有关于字符串和变量连接的内容，我们到字符串再讲

Python 内置数据类型 1

- **字符串**: str
  - 常规字符串, raw 字符串, 三引号 (单, 双三引号)
  - 常用方法: find, count, replace, startswith, endswith, upper, lower, split, join, strip
  - 练习: 将 RNA 序列整理为大写, 并替换为 DNA 序列?
  - 切片 (左闭右开): 常规, 步长, 反向
  - 格式化字符串: %, fstring, format 方法
- **二进制**: bytes
- **数值型**:
  - int
  - float
  - complex
- **序列**:
  - list: [] 新建列表,list 函数, 切片, 更改元素, 常用方法 (append, remove, pop)
  - tuple: (a,) 新建元组,tuple 函数, 切片, 元素不可更改
  - range 对象: 功能, 转 list, 转 tuple, 直接遍历, type
  - 字符串

Python 内置数据类型 2

- **集合**: set:
  - {} 新建集合, set 函数, 常用方法 (add, update, remove , discard)
- **字典**: dict:
  - key: value 新建字典, dict 函数, 访问键值对, 更改键值对中的值, 添加新的键值对, pop 方法弹出值, popitem 方法弹出键值对
- **布尔型**: bool:
  - 定义,bool 函数
  - 大多数值都为 True
    - 如果有某种内容，则几乎所有值都将评估为 True
    - 除空字符串外，任何字符串均为 True
    - 除 0 外，任何数字均为 True
    - 除空列表, 空元组外，任何列表、元组、集合和字典均为 True
    - 对象为 True 或 False 的本质?(__*len*__) 方法返回 0 或 False, 则 bool 函数将其返回为 False

运算符

- 算术运算符
- 赋值运算符
- 比较运算符
- 逻辑运算符
- 身份运算符
- 成员运算符
- 位运算符
- 海象运算符 (:=)

类型转换

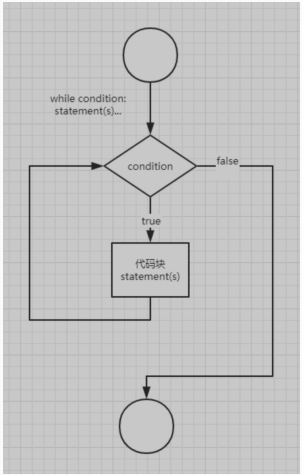
各举一个例子

- **数值**
  - int()
  - float()
  - str()
  - complex()
- **序列**
  - list()
  - tuple()
  - str()
- **集合, 字典 (自己探索)**
  - set()
  - dict()

流程控制

- 顺序结构
- 分支结构——判断
  - if, if else, if elif else
- 循环结构——循环
  - while loop
  - for loop
- 练习：九九乘法表
  - if + for loop

```
1 x 1 =  1
2 x 1 =  2  2 x 2 =  4
3 x 1 =  3  3 x 2 =  6  3 x 3 =  9
4 x 1 =  4  4 x 2 =  8  4 x 3 = 12  4 x 4 = 16
5 x 1 =  5  5 x 2 = 10  5 x 3 = 15  5 x 4 = 20  5 x 5 = 25
6 x 1 =  6  6 x 2 = 12  6 x 3 = 18  6 x 4 = 24  6 x 5 = 30  6 x 6 = 36
7 x 1 =  7  7 x 2 = 14  7 x 3 = 21  7 x 4 = 28  7 x 5 = 35  7 x 6 = 42  7 x 7 = 49
8 x 1 =  8  8 x 2 = 16  8 x 3 = 24  8 x 4 = 32  8 x 5 = 40  8 x 6 = 48  8 x 7 = 56  8 x 8 = 64
9 x 1 =  9  9 x 2 = 18  9 x 3 = 27  9 x 4 = 36  9 x 5 = 45  9 x 6 = 54  9 x 7 = 63  9 x 8 = 72  9 x 9 = 81
```



函数

函数

- **定义:**
  - 函数是一种**仅在调用时运行**的代码块
    - 全局变量 (函数外部创建的变量)
    - 局部变量 (在函数内部被赋值然后使用的变量...)
  - 可以将数据 (参数) 传递到函数中
  - 函数可以把数据作为结果返回
- **种类:**
  - 内置函数 (str(), float(), int()...)
  - 自定义函数
    - 无参数, 返回值
    - 无参数, 有返回值
    - 位置参数
    - 关键字参数
    - 参数数目不确定 *args
    - 参数数目不确定 **kwargs
  - 匿名函数 (不推荐, 不易读)
  - 递归函数 (不易读)
  - 装饰器 (了解, 用到再讲)

```python
# test1
def myfunc():
    x = 1
    return
print(x)
# test2
X = 1
def myfunc2():
    print(X)
    return
myfunc2()
print(X)
# test3
def myfunc3():
    global S
    S = 3
    return
print(S)
myfunc3()
print(S)
```

```python
# test4
def func():
    return

type(func())
a = func()
print(a)
priont(f'{a}')
```

IO 操作

IO 操作-文件打开或创建

在 Python 中使用文件的关键函数是 open() 函数。open() 函数有两个参数：文件名和模式。

- **文件名**
- **模式**
  - 文件打开模式
    - r 读取 - **默认值**。打开文件进行读取，如果文件不存在则报错。
    - a 追加 - 打开供追加的文件，如果不存在则创建该文件。
    - w 写入 - 打开文件进行写入，如果文件不存在则创建该文件。
    - x 创建 - 创建指定的文件，如果文件存在则返回错误。
  - 读作文本或二进制
    - t 文本 - **默认值**。文本模式。
    - b 二进制 - 二进制模式（例如图像）。

IO 操作-文件写入和读取

- open() 函数返回文件对象
- 写入: 文件对象的 write 方法
- 读取: 文件对象的 read 方法用于读取文件的内容
  - read() 默认情况下, read() 方法返回整个文本，也可以指定要返回的字符数
  - readline() readline() 方法每次返回一行 (iterator)
  - readlines() 把文件当做 list 返回, 一行为一个元素

IO 操作-文件关闭和删除

```
1 import os
2
3 os.remove("demofile.txt")  \# 删除文件
4 os.rmdir("myfolder")  \# 删除文件夹
```

语法糖

语法糖 (Syntax Sugar)

在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但更方便程序员使用。简而言之，语法糖让程序更加简洁，有更高的可读性

- **连续比较** 1 < x < 10 (x > 1 and x < 10)
- **三元表达式 (结果一 if 判断条件 else 结果二)**
- **列表推导式**
- **字典推导式**
- **集合推导式**

- **迭代器对象** (Iterator Object)
  - **把一个类作为一个迭代器使用需要在类中实现两个方法** __iter__() **与** __next__()

- **生成器函数** (Generator Function)**(实战项目一会详细演示)(非常重要!)**
  - **在 Python 中，使用了** yield **的函数被称为生成器** (generator).
  - **跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。**
  - **在调用生成器运行的过程中，每次遇到** yield **时函数会暂停并保存当前所有的运行信息，返回** yield **的值, 并在下一次执行** next() **方法时从当前位置继续运行。**
  - **其实** for **循环就在调用** next **方法,next 是一个相对底层的方法,for 循环基于它**
  - **调用一个生成器函数** (Generator Function)**，返回的是一个迭代器对象** (Iterator Object)**。**
- **装饰器 (了解)(实战项目中如有时间空余会演示)**
- ...

Q & A!

Thank you!

第四节 Python 进阶知识

PyCharm 的安装

PyCharm 的安装

- **官网** https://www.jetbrains.com/pycharm/

```
# macos
brew install --cask pycharm-ce  # 社区版
brew install --cask pycharm  # 专业版
# or 直接官网安装，不再掩饰

# windows / linux
# 直接官网安装
# 演示！
```

PyCharm 的配置 (MacOS 为例)

```
# ----------------------------------
# set outer tools
# Auto PEP8 # conda install autopep8
# ----------------------------------
Name: AutoPep8
Description: autopep8 your code
Program: autopep8
Arguments: --in-place --aggressive --aggressive $FilePath$
Working directory: $ProjectFileDir$
Output filters: $FILE_PATH$\:$LINE$\:$COLUMN$\:.*
# ----------------------------------
# plugins recommand
# ----------------------------------
# Key Promoter X
# CodeGlance3
# Json Parser
# Rainbow Brackets
# Rainbow CSV
```

```
# ----------------------------------
# 安装专业版并使用学生(教育)邮箱激活一年
# ----------------------------------
# 注册jetbrains账号
# https://account.jetbrains.com/login
# 使用学生或教师邮箱, 免费申请专业版
# https://www.jetbrains.com/shop/eform/students/
# 然后去邮箱里查看, 按照邮件提示操作即可
# 成功后, 下载安装 PyCharm 专业版本,
# 选择help选项卡 -> registe plugins, 登录账号即可
# ----------------------------------
# 新建一个项目 Project 并运行测试文件
# ----------------------------------
# WSL整合进 PyCharm, 需要专业版
# - 不再演示如何安装 WSL, 如何配置 WSL
# - 查看左下角脚注链接的官方文档
```

---

http://www.zhaohuanan.cc/category/pycharm.html

https://learn.microsoft.com/zh-cn/windows/wsl/

https://www.jetbrains.com/help/pycharm/using-wsl-as-a-remote-interpreter.html#configure-wsl

模块与包

模块与包的概念

- 内建对象 (Built-in Bbjects)
  - print(), str()
  - int, str, float
  - dir(__builtins__)
- 模块 (Module)
  - 需要 import 导入
  - 使用 Python 编写的模块.py
  - 使用 C 编写的动态加载模块 (.dll, .pyd, .so, .sl 等)
  - 内建的 C 编写的模块 sys.builtin_module_names
- 包 (Package)
  - 需要 import 导入
  - 可包含子模块或递归地包含子包的 Python 模块
  - 从技术上说，包是带有 __path__ 属性的 Python 模块。
  - 简单来说，它更像是一个文件夹。
- 标准库 (Standard Library)
  - 标准库相当于解释器的外部扩展，在我们安装 python 解释器的时候，就一块安装上了。
  - 它并不会随着解释器的启动而启动 (builtins)，要想使用这些外部扩展，必须提前导入 (os)。
  - os, sys, time 等这些内置模块的的集合
- 第三模块, 包
- sys.path
  - 自定义模块、包——需要先放到 python 环境变量 (sys.path) 目录下后再导入

演示

- 安装一个第三方包, 如 pandas, matplotlib
  - 导入包, 并演示功能
  - ctr/cmd 鼠标点击包名进入包源代码中
- 自己创建一个自定义模块
  - 导入模块中的函数, 并演示功能
  - __name__ 变量
    - 运行本文件则为 __name__
    - 在别的 py 文件中导入后运行则为模块名!
  - __pycache__
    - python mymodule.cpython-39.pyc

```
1  def myprint():
2      print("我在我的模块里!")
3      print(__name__)
4
5
6  if __name__ == "__main__":
7      myprint()
8  # myprint()
```

异常处理

异常处理

为了保持大型程序的健壮和稳定,python 提供了非常重要的三个功能特性来处理 python 程序在运行中出现的异常和错误。你可以使用该功能来调试 python 程序。

- **异常处理**
  - try except **的关键字**
    - try **后跟正常逻辑**
    - except **捕捉错误信息**
    - else **如果** except **中描述的错误没捕捉到, 就会执行 (可选)**
    - finally **不论是否捕捉到错误, 注定执行 (可选)**
  - import builtins; print(dir(builtins))
    - **官方文档**
    - https://www.runoob.com/python/python-exceptions.html
- **触发异常**
  - raise **函数: Python 允许程序自行引发异常，使用** raise **语句即可**
- **断言** (Assertions)
  - assert True # **条件为** true **正常执行**
  - assert False # **条件为** false **触发异常**
- **演示 (以打开不存在的文件为例)**

面向对象编程

把大象装进冰箱!–函数式编程的问题

```
# 把大象装进冰箱!!
a = "大象"
open_ice_door()  # 开冰箱门，需要自己实现开冰箱门的函数
push(a)   # 推大象进入
close_ice_door()  # 关冰箱门，需要自己实现关冰箱门的函数
```

https://zhuanlan.zhihu.com/p/437925568

把大象装进冰箱!–函数式编程的问题

```python
# 把大象装进冰箱!!
a = "大象"
open_ice_door()  # 开冰箱门，需要自己实现开冰箱门的函数
push(a)   # 推大象进入
close_ice_door()  # 关冰箱门，需要自己实现关冰箱门的函数

# 那如果是把大象装进洗衣机呢？

a = "大象"
open_washer _door()  # 开洗衣机门，需要自己实现开洗衣机门的函数
push(a)   # 推大象进入
close_washer_door()  # 关洗衣机门，需要自己实现关洗衣机门的函数
```

https://zhuanlan.zhihu.com/p/437925568

把大象装进冰箱!–函数式编程的问题

```
# 把大象装进冰箱!!
a = "大象"
open_ice_door()  # 开冰箱门，需要自己实现开冰箱门的函数
push(a)   # 推大象进入
close_ice_door()  # 关冰箱门，需要自己实现关冰箱门的函数

# 那如果是把大象装进洗衣机呢？

a = "大象"
open_washer _door()  # 开洗衣机门，需要自己实现开洗衣机门的函数
push(a)   # 推大象进入
close_washer_door()  # 关洗衣机门，需要自己实现关洗衣机门的函数

# 那如果是把大象装进铁笼呢？

a = "大象"
open_hot_door()  # 开铁笼门，需要自己实现开铁笼门的函数
push(a)   # 推大象进入
close_hot_door()  # 关铁笼门，需要自己实现关铁笼门的函数
```

https://zhuanlan.zhihu.com/p/437925568

万物皆对象

R 是一个函数式编程的语言 (大象装进冰箱的弊端!)

Java 完全面向对象

Python 既可以以函数式编程方式写程序, 也可以以面向对象编程方式写程序!

如果你以前没有接触过面向对象的编程语言, 那你可能需要先了解一些面向对象语言的一些基本特征, 在头脑里头形成一个基本的面向对象的概念, 这样有助于你更容易的学习 Python 的面向对象编程。

接下来我们先来简单的了解下面向对象的一些基本特征。

- **对象?**
  - **几只胳膊 (属性, 名词)**
  - **几条腿 (属性, 名词)**
  - **打人, 吃饭 (方法, 动词)**
- **大象装进? 里面 (冰箱, 洗机器, 铁笼)**
  - **对象都是容器 (属性)**
  - **对象都有 open_door(方法)**
  - **对象都有 close_door(方法)**
  - **对象都有 push(方法)**

面向对象编程思想

- 类 (Class):
  - 用来描述具有相同的属性和方法的对象的集合
    - A, G, C, T这四个碱基 (对象) 都属于碱基这个集合 (类), 都具有分子量属性和互补配对方法
    - 狗, 小猫这两种动物 (对象) 都属于动物这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 南方人, 北方人两种人 (对象) 都属于中国人这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 动物, 人两种生命体 (对象) 都属于生命这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
  - 它定义了该集合中每个对象所共有的属性和方法
- 对象：对象是类的实例; 创建对象的过程叫做实例化
  - 对象包括两个数据成员 (属性) (类变量和实例变量) 和方法
- 属性
  - 类变量：类变量在整个实例化的对象中是公用的
    - 类变量定义在类中且在函数体之外
    - 类变量通常不作为实例变量 (或者说不作为属性) 使用, 只为类的属性和方法的实现服务
  - 实例变量：在类的声明中，属性是用变量来表示的
    - 这种变量就称为实例变量 (属性)，是在类声明的内部但是在类的其他成员方法之外声明的
- 方法：类中定义的函数
  - 方法重写 (继承和多态)
  - 局部变量：定义在方法中的变量，只作用于当前实例的类

面向对象编程思想

- 类 (Class):
  - 用来描述具有相同的属性和方法的对象的集合
    - A, G, C, T这四个碱基 (对象) 都属于碱基这个集合 (类), 都具有分子量属性和互补配对方法
    - 狗, 小猫这两种动物 (对象) 都属于动物这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 南方人, 北方人两种人 (对象) 都属于中国人这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 动物, 人两种生命体 (对象) 都属于生命这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
  - 它定义了该集合中每个对象所共有的属性和方法

面向对象编程思想

- **类 (Class):**
  - 用来描述具有相同的**属性**和**方法**的**对象**的**集合**
    - A, G, C, T这四个碱基 (对象) 都属于碱基这个集合 (类), 都具有**分子量**属性和**互补配对**方法
    - 狗, 小猫这两种动物 (对象) 都属于动物这个集合 (类), 都具有**身高**, **体重**属性和**吃饭**, **睡觉**方法
    - 南方人, 北方人两种人 (对象) 都属于中国人这个集合 (类), 都具有**身高**, **体重**属性和**吃饭**, **睡觉**方法
    - 动物, 人两种生命体 (对象) 都属于生命这个集合 (类), 都具有**身高**, **体重**属性和**吃饭**, **睡觉**方法
  - 它定义了该集合中每个对象所共有的**属性**和**方法**
- **对象：对象是类的实例; 创建对象的过程叫做实例化**
  - 对象包括两个数据成员 (**属性**) (类变量和**实例变量**) 和**方法**

面向对象编程思想

- 类 (Class):
  - 用来描述具有相同的属性和方法的对象的集合
    - A, G, C, T这四个碱基 (对象) 都属于碱基这个集合 (类), 都具有分子量属性和互补配对方法
    - 狗, 小猫这两种动物 (对象) 都属于动物这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 南方人, 北方人两种人 (对象) 都属于中国人这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 动物, 人两种生命体 (对象) 都属于生命这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
  - 它定义了该集合中每个对象所共有的属性和方法
- 对象：对象是类的实例; 创建对象的过程叫做实例化
  - 对象包括两个数据成员 (属性) (类变量和实例变量) 和方法
- 属性
  - 类变量：类变量在整个实例化的对象中是公用的
    - 类变量定义在类中且在函数体之外
    - 类变量通常不作为实例变量 (或者说不作为属性) 使用, 只为类的属性和方法的实现服务
  - 实例变量：在类的声明中，属性是用变量来表示的
    - 这种变量就称为实例变量 (属性)，是在类声明的内部但是在类的其他成员方法之外声明的

面向对象编程思想

- 类 (Class):
  - 用来描述具有相同的属性和方法的对象的集合
    - A, G, C, T这四个碱基 (对象) 都属于碱基这个集合 (类), 都具有分子量属性和互补配对方法
    - 狗, 小猫这两种动物 (对象) 都属于动物这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 南方人, 北方人两种人 (对象) 都属于中国人这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
    - 动物, 人两种生命体 (对象) 都属于生命这个集合 (类), 都具有身高, 体重属性和吃饭, 睡觉方法
  - 它定义了该集合中每个对象所共有的属性和方法
- 对象：对象是类的实例; 创建对象的过程叫做实例化
  - 对象包括两个数据成员 (属性) (类变量和实例变量) 和方法
- 属性
  - 类变量：类变量在整个实例化的对象中是公用的
    - 类变量定义在类中且在函数体之外
    - 类变量通常不作为实例变量 (或者说不作为属性) 使用, 只为类的属性和方法的实现服务
  - 实例变量：在类的声明中，属性是用变量来表示的
    - 这种变量就称为实例变量 (属性)，是在类声明的内部但是在类的其他成员方法之外声明的
- 方法：类中定义的函数
  - 方法重写 (继承和多态)
  - 局部变量：定义在方法中的变量，只作用于当前实例的类

把大象装进冰箱!-OOP, 伪代码

```python
class Box():
    """盒子类，实现了开门、关门方法"""

    def open_door(self):
        pass

    def close_door(self):
        pass

class IceBox(Box):
    """冰箱"""

    def ice(self):
        """制冷"""
        pass

class WaterBox(Box):
    """洗衣机"""

    def add_water(self):
        """加水"""
            pass
```

```python
    def sub_water(self):
        """排水"""
        pass

    def wash(self):
        """洗涤"""
        pass

a = "大象"
ice_box = IceBox()   # 冰箱对象
ice_box.open_door()  # 通知冰箱开门
push(a)   # 推大象进入
ice_box.close_door()  # 通知冰箱关门


# 那我想关老虎呢？
b = "老虎"
ice_box.open_door()  # 通知冰箱开门
push(b)   # 推老虎进入
ice_box.close_door()  # 通知冰箱关门
```

创建类-以碱基互补配对为例

- Class: BioBase(大驼峰命名法)
- 实现 __init__() 方法 (接受碱基类型, 分子量)
- 实现属性
  - 碱基
  - 分子量
- 实现方法
  - 获取碱基
  - 获取分子量
  - 获取互补配对的碱基
  - 设置分子量
- 实例化
- 类变量 (互补配对 dict)
- 实例变量 (分子量)

继承和多态

把大象装进冰箱!-OOP, 伪代码

```python
class Box():
    """盒子类，实现了开门、关门方法"""

    def open_door(self):
        pass

    def close_door(self):
        pass

class IceBox(Box):
    """冰箱"""

    def ice(self):
        """制冷"""
        pass

class WaterBox(Box):
    """洗衣机"""

    def add_water(self):
        """加水"""
            pass
```

```python
    def sub_water(self):
        """排水"""
        pass

    def wash(self):
        """洗涤"""
        pass

a = "大象"
ice_box = IceBox()   # 冰箱对象
ice_box.open_door()  # 通知冰箱开门
push(a)   # 推大象进入
ice_box.close_door()  # 通知冰箱关门


# 那我想关老虎呢？
b = "老虎"
ice_box.open_door()  # 通知冰箱开门
push(b)   # 推老虎进入
ice_box.close_door()  # 通知冰箱关门
```

代码规范

PEP8 规范 (常用的列举)

- **代码布局**
  - 每个缩进级别使用 4 个空格; 连续行使用垂直对齐或者使用悬挂式缩进 (额外的 4 个空格缩进)
  - 空格是首选的缩进方法
  - 每行最多 79 个字符
  - 二元运算符前后换行都允许，只要代码保持一致就行。对于新代码建议在二元运算符前进行换行
  - 空白行：使用两个空白行分隔顶层函数和类定义；类方法定义使用一个空行分隔；使用额外的空白行来分隔相关逻辑功能
  - 文件应该使用 UTF-8 编码, 且不应该有编码声明
  - 导入多个库函数应该分开依次导入；导入总是放在文件的顶部，在任何模块注释和文档字符串之后，在模块全局变量和常量之前；导入应按以下顺序进行：标准库导入、有关的第三方库进口、本地应用程序/库特定的导入, 每组导入直接用空行分隔; 避免通配符导入 (import *)
- **字符串**
  - 单引号字符串和双引号字符串相同，代码保持一致即可
  - 对于三引号字符串，常用三个双引号作文档字符串，文档字符串常用在模块的开端用以说明模块的基本功能，或紧跟函数定义的后面用以说明函数的基本功能
- **空格**
  - 避免使用无关的空格，包括空格内、逗号分号前面等; 避免在行末使用空格
  - 二元运算符在两侧使用一个空格
  - 当用于指示关键字参数或默认参数值时，不要在 = 符号周围使用空格

https://www.cnblogs.com/tangjielin/p/16511066.html

https://peps.python.org/pep-0008/

PEP8 规范 (常用的列举)

- **使用尾部逗号 (trailing commas)**
  - 尾部逗号通常可选, 除了用来说明是只有一个元素的元组 tuple 时
  - 当参数、值等列表期望经常扩展时，通常是每个值一行，再加上一个尾部逗号
- **注释**
  - 代码更改时，相应的注释也要随之高优更改
  - 注释应该是**完整的语句**，第一个单词应该大写，除非它是特定标识符
  - **块注释**：缩进到与该代码相同的级别。块注释的每一行都以 # 和一个空格开始
  - **行注释**：对某一语句行进行注释，注释应该与语句至少隔开**两个空格**，用 # 和一个空格开始
  - 对于公共的 modules, functions, classes, and methods, 需要写文档字符串
  - 注释应该是完整的语句，第一个单词应该大写，除非它是特定标识符
- **命名约定**
  - python 命名规范有点混乱, 很难完全保存一致。对于新模块和包，应该遵守这些新的约定，已存在的库内部一致性更重要
  - 命名应该**反应其用途而非实现**
  - 不要将字符 l(小写字母 l),O(大写字母 o) 或 I(大写字母 I) 作为单个字符变量名称
  - **模块名应该使用简短、全小写**的名字
  - **类的命名采用大驼峰命名法**，即每个单词的首字母大写
  - **函数名称应该是小写的，为了提高可读性，必须使用由下划线分隔的单词**

---

https://www.cnblogs.com/tangjielin/p/16511066.html

https://peps.python.org/pep-0008/

Google Python 命名规范 (常用的列举)

- **命名**
  - 异常名: ExceptionName ;
  - 函数名: function_name ;
  - 全局常量名: GLOBAL_CONSTANT_NAME ;
  - 全局变量名: global_var_name ;
  - 实例名: instance_var_name ;
  - 函数参数名: function_parameter_name ;
  - 局部变量名: local_var_name .
  - 函数名, 变量名和文件名应该是描述性的, 尽量避免缩写, 特别要避免使用非项目人员不清楚难以理解的缩写, 不要通过删除单词中的字母来进行缩写. 始终使用.py 作为文件后缀名, 不要用破折号.
- **命名约定**
  - 所谓”内部 (Internal)”表示仅模块内可用, 或者, 在类内是保护或私有的。
  - 用单下划线 (_) 开头表示模块变量或函数是 protected 的 (使用 from module import 时不会包含)。
  - 用双下划线 (__) 开头的实例变量或方法表示类内私有。
  - 将相关的类和顶级函数放在同一个模块里. 不像 Java, 没必要限制一个类一个模块. 对类名使用大写字母开头的单词 (如 CapWords, 即 Pascal 风格), 但是模块名应该用小写加下划线的方式 (如 lower_with_under.py). 尽管已经有很多现存的模块使用类似于 CapWords.py 这样的命名, 但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致, 这会让人困扰。

https://google.github.io/styleguide/pyguide.html

Google Python 命名规范 (常用的列举)

- ‘__main__’ 和 main()
  - 即使是一个打算被用作脚本的文件, 也应该是可导入的. 并且简单的导入不应该导致这个脚本的主功能 (main functionality) 被执行, 这是一种副作用. 主功能应该放在一个 main() 函数中.
  - 在 Python 中, pydoc 以及单元测试要求模块必须是可导入的. 你的代码应该在执行主程序前总是检查 if __name__ == ‘__main__’ , 这样当模块被导入时主程序就不会被执行.
  - 若使用 absl, 请使用 app.run...所有的顶级代码在模块导入时都会被执行. 要小心不要去调用函数, 创建对象, 或者执行那些不应该在使用 pydoc 时执行的操作.

```python
from absl import app
...
def main(argv):
    # process non-flag arguments
    ...


if __name__ == '__main__':
    app.run(main)

# 否则，使用：
def main():
    ...


if __name__ == '__main__':
    main()
```

https://google.github.io/styleguide/pyguide.html

Google Python 命名规范 (常用的列举)

- **类型注释**
  - 通用规则请先熟悉下 PEP-484 对于方法
  - 仅在必要时才对 self 或 cls 注释 (实战课三当中进行讲解)
  - 若对类型没有任何显示, 请使用 Any
  - 无需注释模块中的所有函数
  - 公共的 API 需要注释
  - 在代码的安全性，清晰性和灵活性上进行权衡是否注释
  - 对于容易出现类型相关的错误的代码进行注释
  - 难以理解的代码请进行注释
  - 若代码中的类型已经稳定，可以进行注释
  - 对于一份成熟的代码，多数情况下，即使注释了所有的函数，也不会丧失太多的灵活性.

https://google.github.io/styleguide/pyguide.html

补充知识

补充知识

- **下划线的 6 个作用**
  - 用在 Python 解释器，表示上一次的执行结果
  - 代码中一个独立的下划线，表示这个变量不重要
  - 类的内部，双下划线作为变量名或函数名的开头，表示私有
  - 双下划线开头和结尾的方法，是魔术方法
    - 比如常见的 '__init__', '__dict__', '__dir__', '__doc__', '__eq__' 等等
  - 作为变量名中间的一部分
  - 作为数字中间的一部分，更易读

补充知识

- Jupyterlab 扔后台?conda/brew/apt install tmux
- 使用服务器的 Jupyterlab
- 服务器 (没有 root 权限) 上安装的 jupyterlab 后自动加载的网页里不能显示 notebook 是啥 (IP 原因)

补充知识

- Windows 配置 scoop 及 scoop 安装软件和命令
  - 直接演示

```powershell
# 进入powershell
set-alias ll  # ls
# 步骤 1:在 PowerShell 中打开远程权限
Set-ExecutionPolicy RemoteSigned -scope CurrentUser;
cd ~
# 步骤 2: 自定义 Scoop 安装目录
# 如果跳过该步骤， Scoop 将默认把所有用户安装的 App 和 Scoop
# 本身置于C:\Users\user_name\scoop
# C:\Users\vagrant
mkdir scoop
$env:SCOOP='C:\Users\vagrant\scoop'
[Environment]::SetEnvironmentVariable('SCOOP', $env:SCOOP, 'User')

# 步骤 3:下载并安装 Scoop
iwr -useb https://gitee.com/glsnames/scoop-installer/raw/master/bin/install.ps1 | iex
scoop config SCOOP_REPO 'https://gitee.com/glsnames/scoop-installer'  # 设置镜像(如果软件安装失败的话)
scoop install git  # 必须的依赖项
```

补充知识

```
# step1: 添加官方维护的extras库(含大量GUI程序)
# 国内源 scoop bucket add extras https://gitee.com/scoop-bucket/extras.git
scoop bucket add extras
# step2 更新源
scoop update
# step3 安装 App和命令行工具(必装工具, conda装好可以不再装了)
scoop install git
scoop install miniconda3  # 安装全完全卸载原来安装的 miniconda!
scoop install sudo  # 调用管理员权限
scooop install tldr  # too long dont read!
# (类 unix 完美,windows 能用但信息提供的是类 unix 的,对 windows 不一定能够完全适用)
# 类unix 可以conda install tldr
scoop install busybox  # 项目实战一会用到! zcat命令的依赖
scoop install cwrsync  # 项目实战一会用到!
scoop install windows-terminal
scoop install powertoys


# 管理:
scoop list  # 查看已安装程序
scoop status  # 查看更新
scoop cleanup  # 删除旧版本
scoop checkup  # 自身诊断
```

Q & A!

Thank you!

项目一　序列文件的处理

序列储存格式的介绍

序列储存格式的介绍–FASTA 文件格式

- FASTA 格式
  - 一种用于表示核苷酸序列或多肽序列的文本格式; 其中碱基对或氨基酸用单个字母来表示
  - 允许在序列前添加序列名及注释
  - 该格式已成为生物信息学领域的一项标准
- FASTA 文件各行记录信息如下:
  - 第一行
    - 由大于号">" 开头的任意文字说明，用于序列标记
    - 为了保证后续分析软件能够区分每条序列，**单个序列的标识必须是唯一的**
  - 第二行
    - 序列本身; 只允许使用既定的核苷酸或氨基酸编码符号。
    - 通常核苷酸符号大小写均可, 而氨基酸常用大写字母。
    - 注意有些程序对大小写有明确要求。一般每行 60-80 个字母。

```
1  >HWI-D00433:463:HNT7JBCXX:1:1101:19071:2193 1:N:0:TTCTCCAT
2  CCTACGGGACGCATCAGTGAGGAATATTGGTCAATGGACGCGAGTCTGAACCAGCCAAG
3  AGCGTGAAGGATGAAGGCCCGATGGGTTGTAAACCTCTTTTATCTGGGAATAAAACGTG
4  CACGTGTGGTATTTTGTATGTACCATAAGAATAAGTATCGGCTAACTCCGTGCCAGCAG
5  CGCGGTAATACGGAGGATCCGAGCGTTATCCGGATTTATTGGGTTTAAAGGGTGCGCAG
6  CGGTCTGTTA
7  >HWI-D00433:463:HNT7JBCXX:1:1101:1713:2316 1:N:0:TTCTCCAT
8  CCTACGGGGTTCACCAGTAGGGAATCTTCCACAATGGGCGAAAGCCTGATTGAGCAAAG
9  CGCGTGTTTTAAGAAGGTCTTCGGATCGTAAAACCCTGTTGTTAGAGAAGAAAGTGCGT
10 CGCGTAACTGTTCACGTTTCTACTGTATCTAACAAGAAAGCACCGGCTAACTACGTTCC
```

https://zhuanlan.zhihu.com/p/363971040

序列储存格式的介绍–FASTA 文件格式

```
# legend server
cd /home/zhaohuanan/3.project/2022_Other_projects/2022-09-25_Prepared_data/FASTA
rsync -avzP rsync://hgdownload.cse.ucsc.edu/goldenPath/mm39/chromosomes/ .
# check一下md5
md5sum -c md5sum.txt

(base) PS C:\Users\vagrant\PythonForBioinformatics> rsync -avzP rsync://hgdownload.cse.
ucsc.edu/goldenPath/mm39/chromosomes/chrM.fa.gz .
receiving incremental file list
chrM.fa.gz
          5,385 100%    5.14MB/s    0:00:00 (xfr#1, to-chk=0/1)

sent 43 bytes  received 5,492 bytes  299.19 bytes/sec
total size is 5,385  speedup is 0.97

# 接下来我们需要将genome生成一下，未确定的基因组区域暂不考虑，只看确定的基因组
# 测试命令
echo `seq 1 1 19` X Y M | awk 'BEGIN {printf "cat "}{for(i=1; i<=NF;i++){printf "chr"$i".fa.gz "}}'
# 写入genome文件
echo `seq 1 1 19` X Y M | awk 'BEGIN {printf "cat "}{for(i=1; i<=NF;i++){printf "chr"$i".fa.gz "}}' | \
    sh > genome_ucsc_mm39.fa.gz
zcat genome_ucsc_mm39.fa.gz | grep -v N | less # linux/window
zcat < genome_ucsc_mm39.fa.gz | grep -v N | less # macos
```

序列储存格式的介绍–FASTQ 文件格式

- **第一行**
  - 以 @ 开头
  - 后面是 reads 的 ID 以及其他信息
  - 例如下一页例中 HWUSI-EAS100R 代表 Illumina 设备名称,
  - 6 代表 flowcell 中的第六个 lane,73 代表第六个 lane 中的第 73 个 tile
  - 941:1973 代表该 read 在该 tile 中的 x:y 坐标信息;
  - #0, 若为多样本的混合作为输入样本, 则该标志代表样本的编号, 用来区分个样本中的 reads/
  - /1 代表 paired end 中的前一个 read。
- **第二行**
  - read 的序列
  - 紧接着下面两行代表该 read 的质量
- **第三行**
  - 以“+”开头, 跟随着该 read 的名称 (一般于 @ 后面的内容相同)，但有时可以省略，但“+”一定不能省

https://baike.baidu.com/item/fastQ 格式 (rmspace)

序列储存格式的介绍–FASTQ 文件格式

- **第四行**
  - 代表 reads 的质量。这一行可以详细说一下!
  - Illumina 测序仪是按照荧光信号来判断所测序的碱基是哪一种的, 例如红黄蓝绿分别对应 ATCG, 那么一旦出现一个紫色的信号该怎么判断呢?
  - 因此对每个结果都有一个概率的问题。起初 sanger 中心用 Phred quality score 来衡量该 read 中每个碱基的质量, 既-10lgP #(lg 意为 log10)
    - 其中 P 代表该碱基被测序错误的概率
    - 如果该碱基测序出错的概率为 0.001, 则 Q 应该为 30, 那么 30+33=63, 那么 63 对应的 ASCii 码为 '?', 则在第四行中该碱基对应的质量代表值即为 '?'
    - ASCII https://baike.baidu.com/item/ASCII

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF

# 例如在NCBI看到的FASTQ格式如下
@HWUSI-EAS100R:6:73:941:1973#0/1
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTT
+HWUSI-EAS100R:6:73:941:1973#0/1
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF
```

序列储存格式的介绍–FASTQ 文件格式

```
# from illumina
@E00591:528:HHVW3CCX2:2:1101:17360:2170 1:N:0:GATCAGCG:ACTA
TGGAGTGAGTACGGTGTGCGTTGAAGTCCTCGTTGTCTTGTTGGCAGGGGTCTGCACCCGGGAGCCCCCGTTCTATATCATCACTGAGATCATGACCTACGGGAACCTCCTCGACTACC
+
JJJJAFFFFJFJJJAFFJJJJFJFFJJFFFJF7A<FJ7JF-<F-7A7<JF7<FJJJ-F<JJJJFJJJJJJ<<AJFJAJJ<7AJFFFJ-<7AFFJJJA-<FF--7FJ-F7-7FA7F<FFJ
# from mgi
@Beta12AdemL1C001R00100000184/1
GTGTCCGTATCTTCACCCCACCACAAACTATTAGCTTTAGAAAGGAAAAGAAAAACCACAACAAAACAGTGTGTGTGTACCAAAGACTTATATGTGCATAAGCAAAAGCAAACAATAGC
+
ICIDIIIDEDIDBIEIIIIEIIEIEEEID@DDE4IDDDEID>EIIEEEEIEEEEEIIEIEEIECEEIE4DIDIDDC6DEIIE@E6EI5CBDEDICIIECEEIIAEEEIIDEEIE:CD:I
# from sra
@fig7_untreated_rep1.1 1 length=150
TTACAAGACTGNTGTATTAGTTTATACTACAAGGACAGGCCCATTNGNNTNTNTTNTTTTNAANTAGGGANATAGTTGGTATTAGGATTAGNATTGTTGTGAAGTATAGTANGGATGCT
+fig7_untreated_rep1.1 1 length=150
AFFFFFFFFFF#FFFFFFFFFFFFFFFFFFFFFFFFFFFF/F#F###F#F#FF#FFFF#FF#FFFFFF#FFFFFFFFFFFFFFFFFFA#FFFFFFFFFFFFFFFFFF#FFFFFF
```

FASTQ 文件的操作

FASTQ 文件的操作

- FASTQ 文件的操作
  - 读取 FASTQ 文件并以 FASTA 格式输出
  - 解析 FASTQ 的质量值, 计算 Q30 比例
  - 根据 FastQC 报告对 FASTQ 文件进行截取
  - 根据 FastQC 报告, 过滤低质量的 lane,tile 数据

FASTA 文件的操作

FASTA 文件的操作

- FASTA 文件的操作
  - 读取 FASTA 文件，并将其中 U 替换成 T
  - 读取 FASTA 文件，并输出反向互补序列
  - 计算基因组序列的长度
  - 计算基因组各染色体的平均 GC 含量
  - 计算基因组中 N 的总长度（effective length）

Q & A!

Thank you!