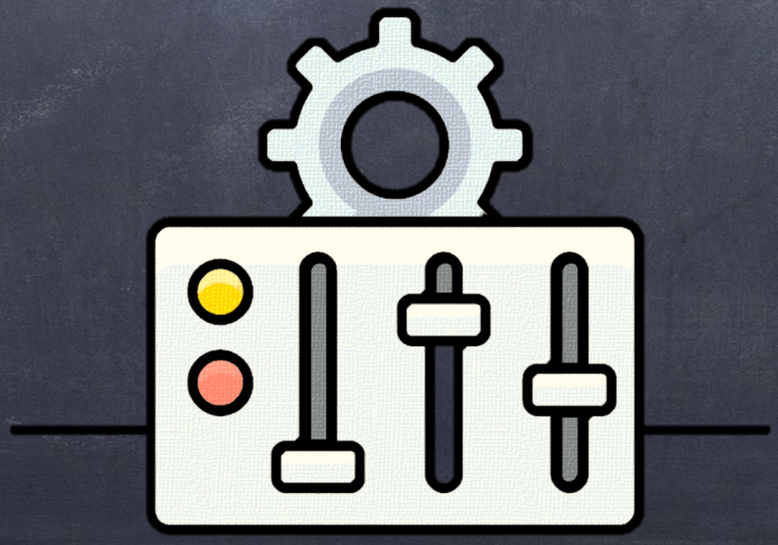
控制



语句

- 之前的赋值、返回语句都是简单语句，其往往就是一行语句。

- 复合（Compound）语句由很多子句（Clause）构成。
- 每个子句以一个头部（header）和冒号开始，包含了之后缩进（indented）的语句序列。
- 比如，Def 就是一个复合语句

注：建议4个空格缩进，而不是Tab。因为这样不同的编辑器下显示会更加一致。
refer to《PEP8》

```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

控制（Control）

目前为止，我们已经有了：

- 表达式（Expression）是为了求值
- 语句（statement）描述了对解释器状态的一些改变，执行语句会应用这些改变。
  - 比如赋值和def语句
- 仅有这些还不能表达复杂的程序，其中一个关键就是"控制"：
  - 某种表达式或者语句，可以控制解释器如何执行程序

条件（Conditional）语句 （If 语句）

语法（syntax）

子句Clause

```
if <condition expression>:
    <suite of statements>
elif <condition expression>:
    <suite of statements>
else:
    <suite of statements>
```

- 以 if 子句开始
- 0个或者多个elif子句
- 0个或者一个else子句，只能在结尾处

语义（Semantics）

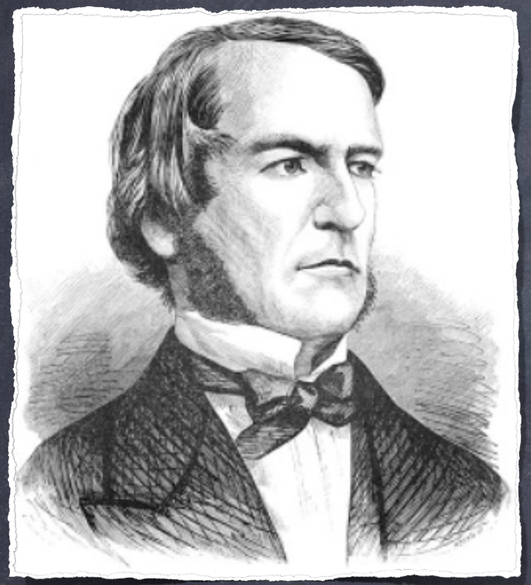
每个子句按照次序考虑：

- 如果该子句头部不是else，那么对该头部的条件表达式进行求值
- 如果值是true,或者头部是else，那么执行该子语的语句序列，忽略并跳过其余头部

条件语句例子

Demo

布尔上下文（Boolean Contexts）



George Boole

两个布尔上下文

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

布尔上下文是在表达式需要被求值为 True 或者False时的地方。

Python中的False值： False, None, 0, “ , [] , (), {}

Python中的True值： 所有其它的值

可以用bool()来判断

布尔表达式（Boolean Expressions）

- 内建的比较运算符可以返回布尔值，即<, >, <=, >=, ==, !=, 在operator模块中有相应的函数。
- python 内建了三个基本的逻辑运算符。
  - and, or, not

布尔表达式（Boolean Expressions）

- <exp1> and <exp2> and <exp3> and …
  - 从前往后求值，直到求到第一个false值，就返回false
  - 如果没有一个false，那么就是最后一个表达式的值True。
- <exp1> or <exp2> or <exp3> …
  - 从前往后求值，直到求到第一个true值，就返回true
  - 如果没有一个true，那么就是最后一个表达式的值false。
- not <exp>
  - 如果<exp>是一个False值，求值结果为True， 如果<exp>是一个True值， 求值结果是False。

短路(Short-Circuiting):
逻辑表达式的真值有时可以不执行全部子表达式而确定

短路（ Short-Circuiting ）

Demo

迭代（Iteration）

- 如果我们编写的每一行代码都只执行一次，程序会变得非常没有生产力。
  - 只有通过语句的重复执行，我们才可以释放计算机的潜力。
- 因此，除了选择要执行的语句，控制语句还用于表达重复操作。

迭代（Iteration）

Demo

Demo

迭代（Iteration）

- While 语句

```
while <expression>:
      <suite of statements>
```

- 语义：
  1. 对头部表达式进行求值。
  2. 如果其为True，那么执行所包含的语句序列，返回步骤1。

斐波拉契数列

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1"""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1              # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

下一个值等于当前值加上前一个值

所有=号右边的求值都发生在绑定之前

Python Tutor

测试

- 有了循环、条件分支、函数，程序逐渐变得复杂
- 我们需要知道我们写的代码是否符合我们的预期
- 测试就是通过执行程序来判断当前程序是否符合预期的一种行为。

测试

- 在Python中，最简单的测试就是doctest
- 利用文档，可以将简单的测试直接附在其中。

```
def sum_naturals(n):
    """Return the sum of the first n natural numbers

    >>> sum_naturals(10)
    55
    >>> sum_naturals(100)
    5050
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total
```

```
python -m doctest <python_source_file>
```

此外，也可直接在程序里执行如下语句：

```
from doctest import run_docstring_examples
run_docstring_examples(sum_naturals, globals())
```

测试

- 一般我们使用断言语句（assert）来判断函数是否正确
- assert <expression>，其中expression是一个布尔表达式，如果其求值为False，则会跳出异常。你可以附加一个字符串来增加该异常的可读性，即assert <expression>, ‘some message’
- 比如：
  assert sum_naturals(100) == 5050, ‘1+2+...+100 = 5050’
- 可以用一个函数专门写多个这样的断言语句来测试某个函数

测试

- 高效测试的关键是在实现新的函数之后（甚至是之前）立即编写（以及执行）测试。
- 只调用一个函数的测试叫做单元测试。
- 详尽的单元测试是良好程序设计的标志。

敏捷开发

Unit Testing

Any questions ?