

## *introduction (week 1)*

*Ben Bolker*

*10:29 12 January 2015*

### *Introduction*

#### *Administrative trivia*

- Instructor: Ben Bolker
  - `bolker@mcmaster.ca`: please include `1mp3` in Subject:
  - `http://www.math.mcmaster.ca/bolker`
  - HH 314 (sometimes LSB 336); office hours TBA
- TA: Jake Szamosi
  - `teaching@jcszamosi.ca`: please allow one full business day for response
- Grading:
  - midterm 20%
  - final (take-home?) 30%
  - weekly assignments 30%
  - project 20%
- Laptop policy
- Course material on [Github](#) and Avenue
- Expectations of professor and students
- Textbook (none); see [resources](#)
- Course content: reasonable balance among
  - nitty-gritty practical programming instruction:

... I just sat down in front of a text editor, with nothing but thoughts, and ended up with a program that did exactly what I wanted it to a few hours later ... ([ankit panda](#))

- – conceptual foundations of computing/computer science
- context/culture of mathematical/scientific computing
- interesting applications

### *Installing Python*

- Everyone must have access to a computer with Python3 installed

- There are detailed instructions for installing Python3 on the [Pragmatic Programming](#) website.
  - \* Click on the ‘Details’ tab and scroll to the appropriate instructions for your operating system.
  - \* On Mac and Linux, once Python3 is installed you can run it directly from the Terminal by typing `python3`. On Windows, go to the Start menu and find Python.
- We are recommending that you use PyCharm to write your programs and test things out.
  - You can find the PyCharm installer [here](#). **The first time you run PyCharm, make sure you choose python 3.4.2 from the drop-down menu**

*More interesting stuff*

*Using computers in math and science*

- math users vs. understanders vs. developers
- develop conjectures; draw pictures; write manuscripts
- mathematical proof (e.g. [four-color theorem](#) and [other examples](#)); computer algebra
- applied math: cryptography, tomography, logistics, finance, fluid dynamics, ...
- applied statistics: bioinformatics, Big Data/analytics, ...
- discrete vs. continuous

*Fun!*

**Hello, world**

```
print('hello, python world!')
```

```
## hello, python world!
```

Python as a fancy calculator:

```
print(62**2*27/5+3)
```

```
## 20760
```

- *reference:* [Python intro section 3.1.1](#)

*Interlude: about Python*

- programming languages
  - Python: [scripting](#); high-level; glue; general-purpose; flexible
  - contrast: *domain-specific* scripting languages (MATLAB, R, Mathematica, Maple)
  - contrast: *general-purpose* scripting languages (Perl, PHP)
  - contrast: general-purpose *compiled* languages (Java, C, C++) (“close to the metal”)
- relatively modern (1990s; Python 3, 2008)
- currently the [8th most popular computer language](#) overall; [most popular for teaching](#)
- well suited to mathematical and scientific programming ([NumPy](#); [SciPy](#))
- ex.: [Sage](#); [BioPython](#)

**the Mandelbrot set**

Suppose we iterate  $z_{n+1} = z_n^2 + c$ , for some complex number  $c$ , starting with  $z_0 = 0$ . The [Mandelbrot set](#) is the set for which the iterations do *not* go off to infinity. (*What happens for  $c = 0$ ?  $c = -1$ ?  $c = i$ ?  $c = 1$ ?*)

We can iterate by hand ...

```
print(complex(0,0.65)**2+complex(0,0.65))
print((complex(0,0.65)**2+complex(0,0.65))**2+complex(0,0.65))
print(((complex(0,0.65)**2+complex(0,0.65))**2+complex(0,0.65))**2+complex(0,0.65))

## (-0.4225+0.65j)
## (-0.24399375+0.10075j)
## (0.0493823875391+0.600835259375j)
```

Use **assignments** to simplify ...

```
z0=0
c=complex(0,0.65)
z1=z0**2+c
z2=z1**2+c
z3=z2**2+c
print(abs(z3)<2)

## True
```

The basic method for generating pretty pictures is:

- for lots of different values of  $c$ 
  - set  $z_0 = 0$
  - keep calculating  $z_{n+1} = z_n^2 + c$  until  $\text{mod}(z_{n+1})$  is greater than 2
  - record the final value of  $n$
- translate values of  $n$  into some colour scale and plot the results

Complex arithmetic is built into Python


(*What is  $(2 + 3i)^2 = (\text{complex}(2,3))^{**2}$ ?*)

[Mandelbrot set program](#)

**Note:**

- easier to understand/modify than write from scratch
- build on existing components (*modules*)

### *Interfaces*

- command line/console (PyCharm: **View/Tool Windows/Python Console**)
- programming editor
- integrated development environment (IDE)
- **not** MS Word! 

### **Features**

- syntax highlighting
- bracket-matching
- hot-pasting
- integrated help
- integrated debugging tools
- integrated project management tools
- **most important:** maintain reproducibility; well-defined **work-flows**

## Assignment

- superficially simple
  - = is the **assignment operator**
  - <variable>=<value>
  - variable names
    - \* what is legal? (letters, numbers, underscores, start with a letter)
    - \* what is customary? [convention](#) is `variables_like_this`
    - \* what works well? `v` vs. `temporary_variable_for_loop`
    - \* parallels with file names; directory/folder organization
- variables can be of different **types**
  - built-in: integer (`int`), floating-point (`float`), complex, **Boolean** (`bool`: `True` or `False`),
  - *dynamic* typing
  - (relatively) *strong* typing
    - \* try `print(type(x))` for different possibilities (`x=3`; `x=3.0`; `x="a"`)
    - \* *what happens if you try `x=a`?*
    - \* **don't be afraid to experiment!**

```
x=3
y=3.0
z="a"
q=complex(1,2)
type(x+y)  ## mixed arithmetic
type(int(x+y))  ## int(), float() convert explicitly
type(x+z)
type(q)
type(x+q)
type(True)
type(True+1)  ## WAT
```

## Comparisons and logical expressions

- comparison: (`==`, `!=`)
- inequalities: `>`, `<`, `>=`, `<=`,
- basic logic: (`and`, `or`, `not`)
- remember your truth tables, e.g. `not(a and b)` equals `(not a) or (not b)`

```

a = True; b = False; c=1; d=0
a and b
not(a and not b)
a and not(b>c)
a==c  ## careful!
not(d)
not(c)

```

- **operator precedence:** same issue as [order of operations in arithmetic](#); **not** has higher precedence than **and**, **or**. When in doubt use parentheses ...

From [CodingBat](#):

We have two monkeys, a and b, and the parameters `a_smile` and `b_smile` indicate if each is smiling. We are in trouble if they are both smiling or if neither of them is smiling. Return `True` if we are in trouble.

```

monkey_trouble(True, True) → True
monkey_trouble(False, False) → True
monkey_trouble(True, False) → False

```

### *String operations*

reference: [Python intro](#) section 3.1.2

- Less generally important, but fun
- `+` concatenates
- `*` replicates and concatenates
- `in` searches for a substring

```

a = "xyz"
b = "abc"
a+1  ## error
a+b
b*3
(a+" ")*5
b in a

```

From [CodingBat](#):

Given two strings, a and b, return the result of putting them together in the order abba, e.g. “Hi” and “Bye” returns “HiByeByeHi”.

*Regular expressions*

Large topic – somewhat more advanced than ‘basic programming’, but worth a digression.

What if we are looking for some number, but we don’t know what number?

```
import re
bool(re.search('[0-9]', 'Plan 9'))
```

Pattern	Description
<code>^</code>	Beginning of line
<code>\$</code>	End of line
<code>.</code>	Any single character except newline
<code>[...]</code>	Any single character in brackets
<code>[^...]</code>	Any single character <b>not</b> in brackets
<code>re*</code>	0 or more occurrences of preceding expression
<code>re+</code>	1 or more occurrence of preceding expression
<code>re?</code>	0 or 1 occurrence of preceding expression
<code>re1 re2</code>	match <code>re1</code> or <code>re2</code>
<code>()</code>	grouping

- How would you test whether a string contains a numeric value at the end (e.g. “Plan 99”)?
- What if the string might contain a comma (e.g. “Plan 99,478”)?
- What if you’re looking for the abbreviations of rooms in Hamilton Hall (my office is HH314)?
- ... rooms in LSB *or* HH?

*Lists and indexing*

reference: [Python intro](#) section 3.1.3

*Lists*

- Use square brackets `[]` to set up a **list**
- Lists can contain anything but usually homogeneous
- Put other variables into lists

- Put lists into lists! (“yo dawg ...”)
- `range()` makes a **range** but you can turn it into a list with `list()`
- *Set up a list that runs from 101 to 200*
- *Make a list that ...*

## Indexing and slicing

### Indexing

- Extracting elements is called **indexing** a list
- Indexing **starts from zero**
- Negative indices count backward from the end of the string (-1 is the last element)
- Indexing a non-existent element gives an error

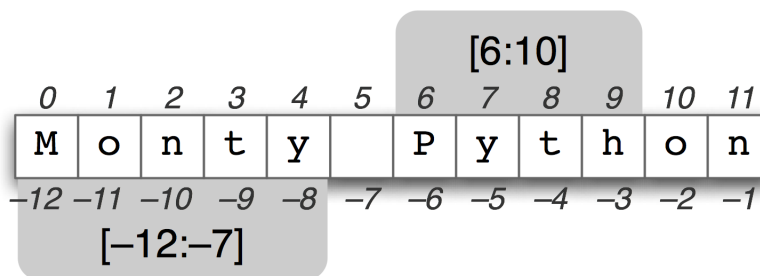


Figure 1: slicing

### Slicing

- Extracting (consecutive) sets of elements is called **slicing**
- Slicing non-existent element(s) gives a truncated result
- Slicing specifies *start*, *end*, *step* (or “stride”)
- Leaving out a bit goes from the beginning/to the end
- Slicing works on strings too!

```

x[:]      # everything
x[a:b]    # element a (zero-indexed) to b-1
x[a:]     # a to end
x[:b]     # beginning to b
x[a:b:n]  # from a to b-1 in steps of n

```

- generate a list of odd numbers from 3 to 15
- reverse a string?



## Other list operations

- Lots of things you can do with lists!
- Lists are **mutable**

```
x = [1,2,3]
y = x
y[2] = 17
print(x)
```

```
## [1, 2, 17]
```

- *operators* vs. *functions* vs. *methods* `x+y` vs. `foo(x,y)` vs. `x.foo(y)`
  - list *methods*
  - appending and extending:

```
x = [1,2,3]
y = [4,5]
x.append(y)
print(x)
```

```
## [1, 2, 3, [4, 5]]
```

```
x = [1,2,3] # reset x
y = [4,5]
x.extend(y)
print(x)
```

```
## [1, 2, 3, 4, 5]
```

Can use `+` and `+=` as shortcut for extending:

```
x = [1,2,3]
y = [4,5]
z = x+y
print(z)
```

```
## [1, 2, 3, 4, 5]
```

- `x.insert(position,value)`: inserts (or `x=x[0:position]+[value]+x[position+1:len(x)]`)
- `x.remove(value)`: removes *first* value
- `x.pop(position)` (or `del x[position]` or `x=x[0:position]+x[position+1:len(x)]`)

- `x.reverse()` (or `x[::-1]`)
- `x.sort()`: what it says
- `x.count(value)`: number of occurrences of `value`
- `x.index(value)`: first occurrence of `value`
- `value in x`: does `value` occur in `x`? (or `logical(x.count(value)==0)`)
- `len(x)`: length

**Note:** [pythonicity](#) vs. [TMTOWTDI](#)