Intro to Python (Class 2)

Ben Bettisworth

1. Values

2. Variables

3. Functions

4. Scope

Section 1

Values

Values

- As we saw in the previous class, programs operate on the state of the computer to produce a new state
- The collections of memory that a program operates on are called “Values”.
  - `3` is a value
  - `3.14` is a value
  - `"foo"` is a value
  - $1 + 2i$ is a value
  - `None` is a value

Operations on Values

- Each instruction in a program can be thought of as “take these values, and perform some operation to them”
- Examples
  - `3 + 2` means take the values 3 and 2 and add them together
  - `3.14 * 2` means take the value 3.14 and double it.
  - `"ice" + "cream"` means take the values `ice` and `cream` and concatenate them to make `icecream`
  - `(1 + 2j) * (1 - 2j)` means take the values $1 + 2i$ and $1 - 2i$ and multiply them

Types

- You might have noticed in the previous examples that + is used in both:
  - `3 + 2`, and
  - `"ice" + "cream"`.
- But these are not the same operation!
- How does python know what to do?

Types

- Values in python have a “Type”, which indicates what operations are valid, and what those operations do.
- Examples of types are:
  - Integer (`3`)
  - Float (`3.14`)
  - Complex Number (`1+1j`)
  - String (`"foo"`)
  - Boolean(`True`)
  - Null value (`None`)

Exercise

Try the following operations in python:

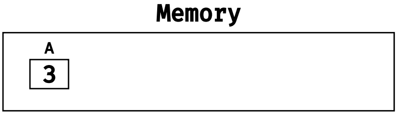
- `"foo" * 2`
- `3 / 2`
- `3 // 2`
- `3 % 2`
- `3 ** 2`

Can you think of other operations that would make sense? Try them!

Section 2

Variables

Variables



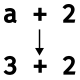
Memory

A

3

- A variable is a “box” that a value is stored in.
- For example, `a = 3` stores the value 3 in a box with the label `a`

Variables


a + 2

3 + 2

- When used in place of a value, a variable acts as if it has the value in it’s box
- `a + 2` is the same as `3 + 2`

Variables

- In python, variables are created with the "assignment operator" =
- `a = 3` or `foo = "bar"`
- Variables name have some requirements:
  - Must start with a letter;
  - Must not contain punctuation (., +, /, %, -, etc.)
    - Exception is _, which is allowed;
  - Can't be a keyword (e.g. `for`, `None`, etc.);
  - And no spaces.

Variable Name Examples

- `a`
- `foo`
- `hereIsACleverName`
- `biology_rules`
- `physics_drools`
- `a1`
- `x2x`

Assignment

- Properties of assignment
    - Transitive `a = b = c`
    - non-associative `a = b; b = c` is not the same as `b = c; a = b`
    - non-communicative `a = b` is not the same as `b = a` (same for `a=b=c`)

Exercise

Demonstrate that the following properties hold for assignment (Write an example):

- Transitivity (`a = b = c`)

And demonstrate that the following properties do *not* hold:

- Associativity (`a = (b = c)`)
    - But compare (`a = (b := c)`)
- Communication (`a = b = c` vs `b = a = c`)

Section 3

Functions

Review Exercise

Problem

Compute the roots of the polynomial

$$4x^2 + 5x - 2$$

Using the Quadratic formula

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Review Exercise

(A Possible) Solution

```
a = 4
b = 5
c = -2

tmp = (b**2 - 4 * a * c)**0.5

root1 = (-b + tmp) / (2 * a)
root2 = (-b - tmp) / (2 * a)

print(root1,  root2)
```

Output

```
0.3187293044088437 -1.5687293044088437
```

Functions

- What if we had two polynomials?
- What if we had four polynomials?
- What if we had a thousand?

Example

```
a, b, c = 4, 5, -2
tmp = (b**2 - 4 * a * c)**0.5
root1 = (-b + tmp) / (2 * a)
root2 = (-b - tmp) / (2 * a)
print(root1,  root2)

a, b, c = 2, 7, 20
tmp = (b**2 - 4 * a * c)**0.5
root1 = (-b + tmp) / (2 * a)
root2 = (-b - tmp) / (2 * a)
print(root1,  root2)
```

Example

Instead we could write a function:

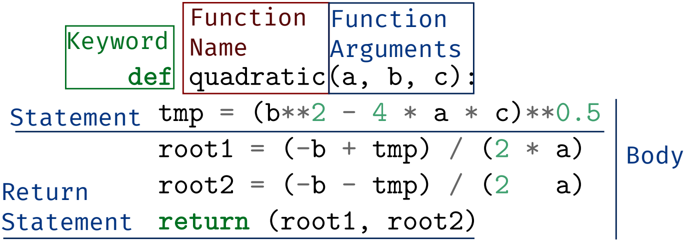
Quadratic Formula as a function

```
def quadratic(a, b, c):
  tmp = (b**2 - 4 * a * c)**0.5
  root1 = (-b + tmp) / (2 * a)
  root2 = (-b - tmp) / (2 * a)
  return (root1, root2)
```

Calling the function

```
quadratic(4,5,-2)
> (0.3187293044088437, -1.5687293044088437)
```

Anatomy of a function



Keyword

Function Name

Function Arguments

```
def quadratic(a, b, c):
    tmp = (b**2 - 4 * a * c)**0.5
    root1 = (-b + tmp) / (2 * a)
    root2 = (-b - tmp) / (2   a)
    return (root1, root2)
```

Statement

Return Statement

Body

Writing a function

First line

```
def foo(bar):
```

Syntax

- Every function starts with `def`
- Then the function name
    - Rules are the same as variables
- Then a parenthesis enclosed list
    - `(a,b)`
    - also `()`
- Finally a ":"{.python}

Writing a function

Next Lines

```
def foo(bar):
  tmp = bar ** 2
  return tmp
```

Syntax

- After the first line of a function, lines are indented
- Python uses indentation to know when the function ends

Writing a function

Calling a function

```
def foo(bar):
  tmp = bar ** 2
  return tmp

print(foo(2))
```

Result

```
4
```

Exercise

Write a function that computes the volume of a sphere using the formula

$$V = \frac{4}{3}\pi r^3$$

Where the only argument is r. You can take $\pi = 3.14$.

Exercise

(A Possible) Solution

```
def sphere_v(r):
  return (4 / 3) * 3.14 * r ** 3

print(sphere_v(3))
```

Output

```
113.03999999999999
```

Exercise

Write a function that computes the probability of rolling a single 1 on *n* *k*-sided dice, with *n* and *k* as function arguments.

Exercise

(A Possible) Solution

```python
def prob_one_k(n,k):
  prob_1 = 1 / k
  prob_n1 = ((k-1)/k) ** (n-1)
  return prob_1 * prob_n1 * n

print(prob_one_k(1, 6))
print(prob_one_k(4, 6))
```

Output

```
0.16666666666666666
0.3858024691358025
```

Wrapping up

What is a function?

- A collection of statements which are executed together.
- A way of organizing code.

Section 4

Scope

Scope

In the following code, what will happen?

```
def make_fav_food():
  fav_food = "olive"

make_fav_food()
print(fav_food)
```

Scope

Code

```python
def make_fav_food():
  fav_food = "olive"

make_fav_food()
print(fav_food)
```

Result

```
Traceback (most recent call last):
  File "<python-input-0>", line 5, in <module>
    print(fav_food)
          ^^^^^^^^
NameError: name 'fav_food' is not defined
```

Scope

Variables in python are *scoped*, which means they are only valid for a specific context.

Example

```
scope_1 = "this is valid in scope 1"
def make_scope():
  scope_2 = "this is valid in scope 2"
```

Nested Scope

Example

```
scope_1 = "this is valid in scope 1"
def make_scope():
  scope_2 = "this is valid in scope 2"
  print(scope_1, scope_2)

make_scope()
print(scope_1, scope_2)
```

Nested Scope

Output

```
this is valid in scope 1 this is valid in scope 2

Traceback (most recent call last):
  File "<python-input-2>", line 7, in <module>
    print(scope_1, scope_2)
                   ^^^^^^^
NameError: name 'scope_2' is not defined.
  Did you mean: 'scope_1'?
```

Creating a New Scope

Scopes are created in: - Modules (source files) - Functions - Classes

Exercise

What does this code print?

```
bar = "hello :)"
def foo():
  bar = "goodbye ;)"

foo()
print(bar)
```

Exercise

What does this code print?

```
bar = "hello :)"
def foo():
    bar = "goodbye ;)"

foo()
print(bar)
```

How do you make it print “goodbye ;)”?

Exercise

What does the following code print?

```
def foo(a):
  a = "howdy!"

b = "See ya!"
foo(b)
print(b)
```

Exercise

Code

```
def foo(a):
  a = "howdy!"

b = "See ya!"
foo(b)
print(b)
```

Output

```
See ya!
```

Methods

You might have noticed that `a.append(1)` has a new kind of syntax.

- `append` is a *method*
- Methods are like functions, but they operate on values.
- E.g. `append` modifies the list `a`.
- Some methods don't modify the value
  - `count(x)`

Methods

Summary

You should think of methods as acting on the value which they are called on.

```
a.append(1)
```

Here, `append` is called on `a`, and adds the value 1 to the end of `a`.