ECE 20875
Python for Data Science

Chris Brinton and David Inouye

(Adapted from material developed by
Prof. Milind Kulkarni and Prof. Chris Brinton)

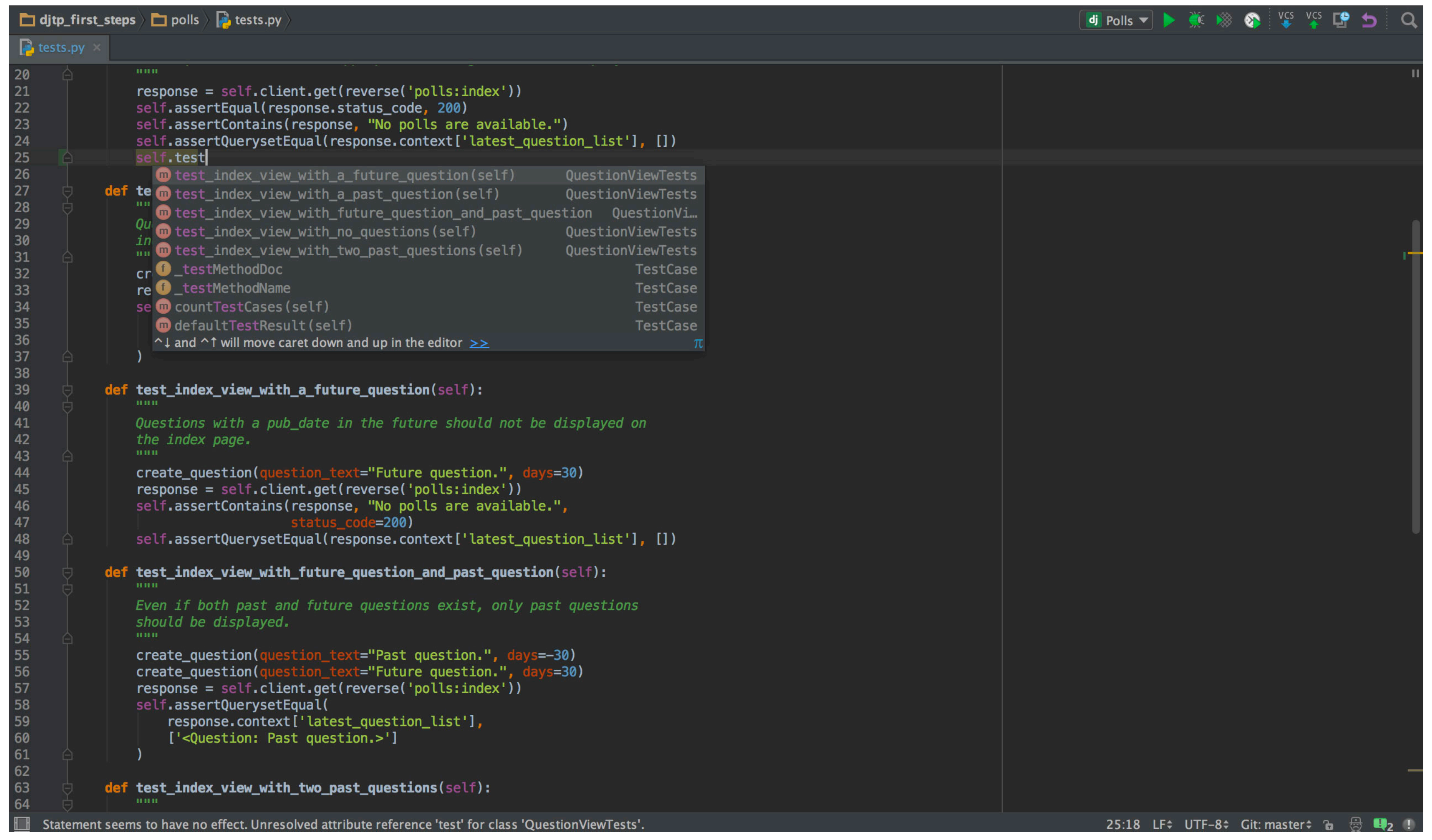
MWF, 12:30pm-1:20pm

Section 1: WALC 1055
Section II: FRNY G124

python basics

coding in python

- Standard Integrated Development Environments (IDEs)
  - IDLE: Python's own, basic IDE
  - PyCharm: Code completion, unit tests, integration with git, many advanced development features (https://www.jetbrains.com/pycharm/)
  - Many more!
- Jupyter Notebook (https://jupyter.org/)
  - Contains both computer code and rich text elements (paragraphs, figures, …)
  - Supports several dozen programming languages
  - Very useful for data science development!
  - You can download the notebook app or use Jupyter Hub available on RCAC (https://www.rcac.purdue.edu/compute/scholar)

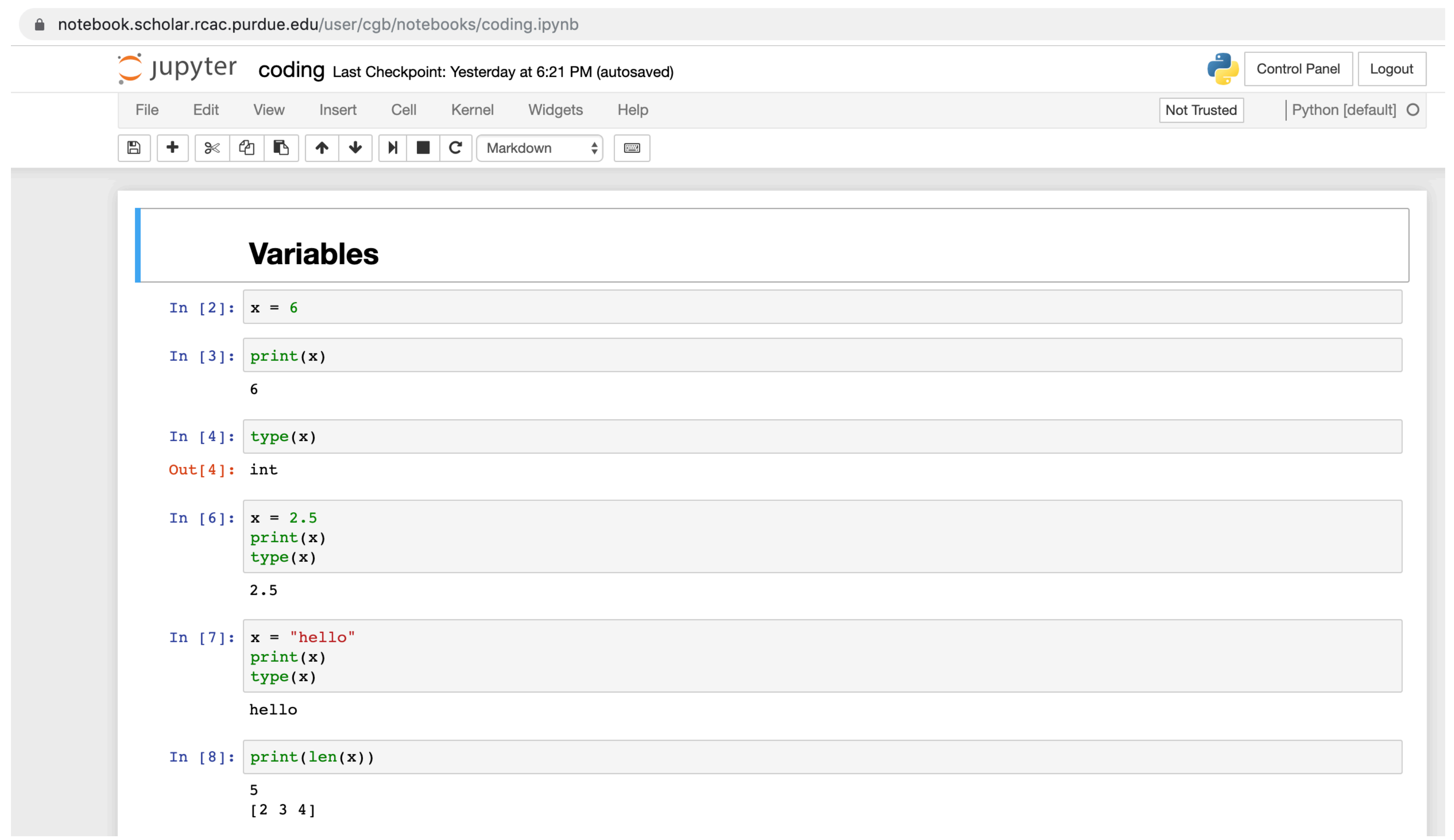


```
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])
        self.test
         test_index_view_with_a_future_question(self)              QuestionViewTests
    def te  test_index_view_with_a_past_question(self)             QuestionViewTests
        "" test_index_view_with_future_question_and_past_question  QuestionVi…
        Qu test_index_view_with_no_questions(self)                 QuestionViewTests
        in test_index_view_with_two_past_questions(self)           QuestionViewTests
        "" _testMethodDoc                                          TestCase
        cr _testMethodName                                         TestCase
        re countTestCases(self)                                    TestCase
        se defaultTestResult(self)                                 TestCase
           ^↓ and ^↑ will move caret down and up in the editor >>
        )

    def test_index_view_with_a_future_question(self):
        """
        Questions with a pub_date in the future should not be displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.",
                            status_code=200)
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        should be displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_two_past_questions(self):
        """
Statement seems to have no effect. Unresolved attribute reference 'test' for class 'QuestionViewTests'.   25:18  LF÷  UTF-8÷  Git: master÷
```


notebook.scholar.rcac.purdue.edu/user/cgb/notebooks/coding.ipynb
jupyter
coding Last Checkpoint: Yesterday at 6:21 PM (autosaved)
Control Panel
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python [default]
Markdown

Variables

```
In [2]: x = 6

In [3]: print(x)

        6

In [4]: type(x)

Out[4]: int

In [6]: x = 2.5
        print(x)
        type(x)

        2.5

In [7]: x = "hello"
        print(x)
        type(x)

        hello

In [8]: print(len(x))

        5
        [2 3 4]
```

basic variables

- No “declaration” command as in other programming languages
    - Variable is created when a value is assigned to it
    - Can change type after they have been set
- Few rules on naming: Can make them very descriptive!
    - Must start with a letter or underscore
    - Case-sensitive (purdue & Purdue are different)
- Combinations (+) work on all types

```
“xyz ” + “abc” = “xyz abc”

3.2 + 1 = 4.2
```

operators and control statements

- Comparison operators:

```
a == b, a != b, a < b,

a <= b, a > b, a >= b
```

- If statement:

```
if r < 3:
  print("x")
```

- If, elif, else (multiline blocks):

```
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b”)
```

- Arithmetic operators:

```
a + b, a - b, a * b,

a / b, a % b, a ** b
```

- Assignment operators:

```
a = b, a += b, a -= b,

a *= b, a /= b, a **= b
```

- Logical operators:

```
(a and b), (a or b),

not(a), not(a or b)
```

lists

- One of the four collection data types
    - Also tuples, sets, and dictionaries
- Lists are ordered, changeable, and allow duplicate members

```
thislist =
["apple", "banana", “apple”,
“cherry”]
```

- Can pass in an integer index, or a range of indexes

```
thislist[0] = “apple"
thislist[-1] = “cherry”
thislist[1:3] = [“banana”, “apple”]
```

- Length using len() method

```
print(len(thislist))
```

- Adding items to a list

```
thislist.append(“orange”)
thislist.insert(1, “orange”)
```

- Removing items from a list

```
thislist.remove(“banana”)
thislist.pop(1)
```

- Defining lists with shorthand

```
new_list = 5 * [0]

new_list = range(5)
```

loops (more control statements)

- while loop: Execute while condition is true

```
i = 1
while i < 6:
  print(i)
  i += 1
```

- for loop: Iterate over a sequence

```
for x in "banana":
  print(x)
```

- `range()` operator can be a useful loop iterator:

```
for x in range(5,10):
   y = x % 2
   print(y)
```

- `break`: Stop a loop where it is and exit
- `continue`: Move to next iteration of loop

```
for val in “sammy_the_dog”:
   if val == “h":
      break
   print(val)
```

lists in for loops

- In other programming languages, for loop variables are integers
- In Python, can use any ‘iterable’ object

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

- Nested loops can be used too

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

- Can also iterate through a list of lists

```
data_list = [[1,2],[2,6],[5,7]]
for point in data_list:
   [x,y] = point
   z = x ** 2
   print(x,y,z)
```

- Can use the range function to iterate through integers

```
for x in range(2, 30, 3):
  print(x)
```

- Can use a list to index another list

```
ind = [1, 3, 5, 7]
values = [0] * 8
for i in ind:
  values[i] = i / 2
```

functions

- Block of code which runs when called
- Defined using def keyword

```
def my_function():
  print("Hello from a function”)
```

- Call a function using its name

```
my_function()
```

- Parameters can be passed as input to functions

```
def my_function(country):
  print("I am from " + country)
```

- To return a value, use the return statement

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
```

- For multiple arguments, can use keywords to specify order

```
def arithmetic(x,y,z):
  return (x+y)/z

print(arithmetic(z=3,x=2,y=4))
```

tuples

- Another of the four collection data types
- Tuples are ordered, **un**changeable, and allow duplicate members

```
thistuple = (“apple", "banana", “apple”, 
“cherry”)
```

- Indexed the same way as lists

```
thistuple[0] = “apple"
thistuple[-1] = “cherry”
thistuple[1:3] = (“banana”, “apple”)
```

- Once a tuple is created, items cannot be added or changed
    - Workaround: Change to list, back to tuple

- One “exception”: If a tuple contains a reference to something changeable, that *something* can be changed
- Check if item exists

```
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits 
tuple")
```

- Tuple with one item needs comma

```
thistuple = (“apple",) #Tuple
thistuple = (“apple") #Not a tuple
```

- Built in functions

```
thistuple.count(“apple")
thistuple.index(“apple")
```

sets

- Collection which is **un**ordered, (half) changeable, and does **not** allow duplicates

- Written with curly brackets

```
thisset = {“apple”, "banana",
“cherry”}
```

- Cannot access items by index, but can loop through and check for items

```
for x in thisset:
  print(x)

print("banana" in thisset)
```

- Cannot change existing items, but can add and remove items

```
thisset.add(“orange")
thisset.update(["orange", "mango", “gra
pes"])
thisset.remove("banana")
```

- Also have set operations just like mathematical objects

```
set1 = {"a", "b", "c"}
set2 = {1, "b", 3}

set1.union(set2)  #Union
set1.intersection(set2)  #Intersection
set1.difference(set2)  #set1 \ set2
set1.issubset(set2)  #Testing if subset
```

dictionaries

- Collection which is **un**ordered, changeable, and indexed
- Also written with curly brackets, but have keys and values

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

- Access/change/add values of items by referring to the key name

```
thisdict[“model"]
thisdict[“year"] = 2019
thisdict[“color”] = "red"
```

- Can iterate through the keys, values, or both

```
for x in thisdict:
  print(thisdict[x])
for x in thisdict.values():
  print(x)
for x, y in thisdict.items():
  print(x, y)
```

- Like other collections, can create a dictionary of dictionaries

```
child1 = {"name" : “Emil", "year" : 2004}
child2 = {"name" : “Tobias", "year" : 2007}
child3 = {"name" : “Linus", "year" : 2011}

myfamily = {“child1" : child1, "child2" : child2,
"child3" : child3}
```

- Use the copy method (not direct assignment) to make a copy of a dictionary

```
mydict = thisdict.copy()
```