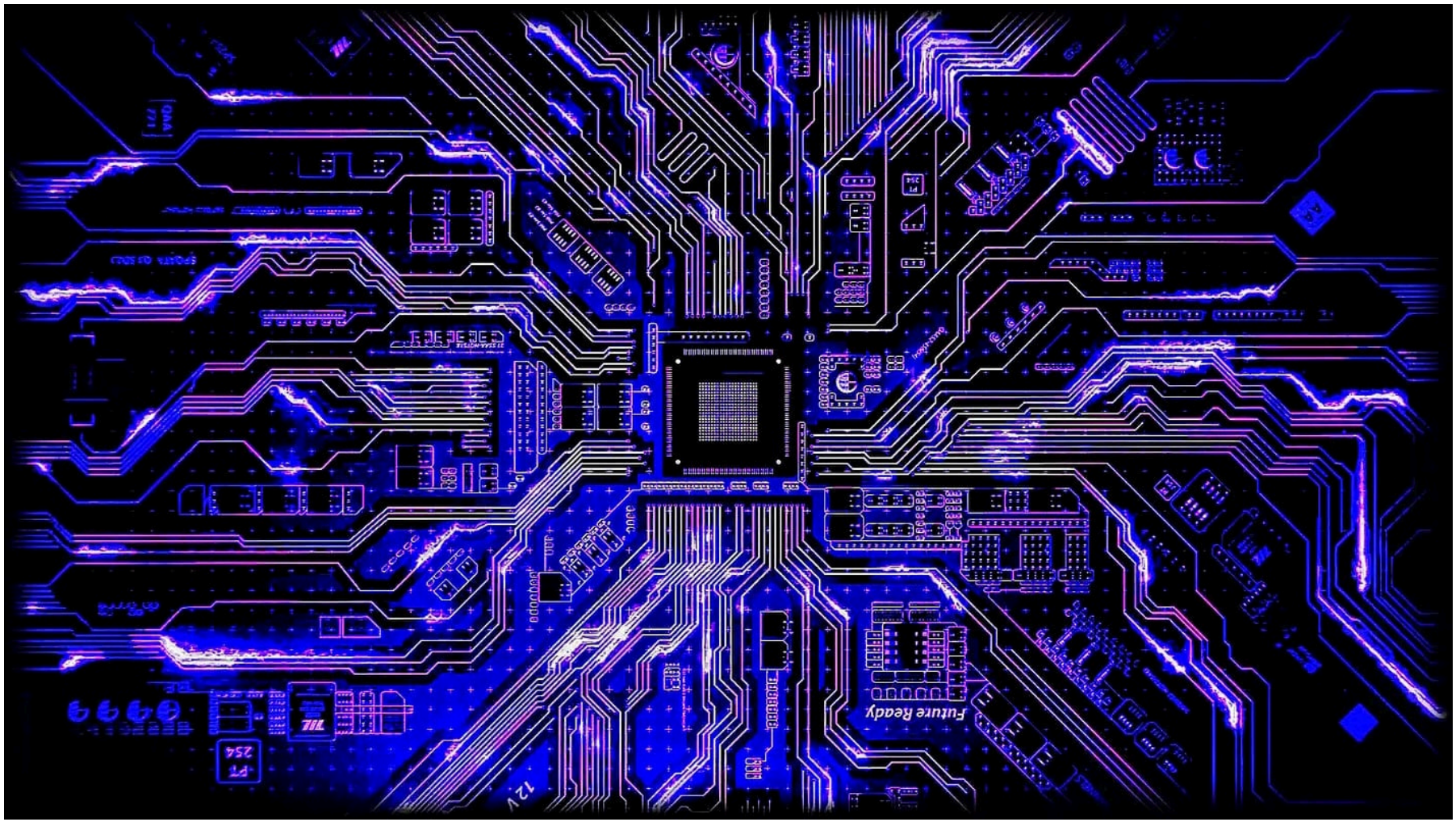
CSci 127: Introduction to Computer Science



hunter.cuny.edu/csci

Frequently Asked Questions

From email and tutoring.

- **When is the final? Is there a review sheet?**

Frequently Asked Questions

From email and tutoring.

- **When is the final? Is there a review sheet?**
  The official final is the last day ***(JULY 11) from 10am - 12pm.***

Frequently Asked Questions

From email and tutoring.

- **When is the final? Is there a review sheet?**
  The official final is the last day ***(JULY 11) from 10am - 12pm.***
  Instead of a review sheet, we have:

Frequently Asked Questions

From email and tutoring.

- **When is the final? Is there a review sheet?**
  The official final is the last day ***(JULY 11) from 10am - 12pm.***
  Instead of a review sheet, we have:
  - *All previous final exams (and answer keys) on the website.*

Today's Topics

- Design Patterns: Searching
- Python Recap
- Machine Language
- Machine Language: Jumps & Loops
- Binary & Hex Arithmetic
- Final Exam: Format

Today's Topics

- **Design Patterns: Searching**
- Python Recap
- Machine Language
- Machine Language: Jumps & Loops
- Binary & Hex Arithmetic
- Final Exam: Format

Predict what the code will do:

```
def search(nums, locate):
    found = False
    i = 0
    while not found and i < len(nums):
        print(nums[i])
        if locate == nums[i]:
            found = True
        else:
            i = i+1
    return(found)

nums= [1,4,10,6,5,42,9,8,12]
if search(nums,6):
    print('Found it! 6 is in the list!')
else:
    print('Did not find 6 in the list.')
```

Python Tutor

```
def search(nums, locate):
    found = False
    i = 0
    while not found and i < len(nums):
        print(nums[i])
        if locate == nums[i]:
            found = True
        else:
            i = i+1
    return(found)

nums= [1,4,10,6,5,42,9,8,12]
if search(nums,6):
    print('Found it! 6 is in the list!')
else:
    print('Did not find 6 in the list.')
```

(Demo with pythonTutor)

Design Pattern: Linear Search

```
def search(nums, locate):
    found = False
    i = 0
    while not found and i < len(nums):
        print(nums[i])
        if locate == nums[i]:
            found = True
        else:
            i = i+1
    return(found)

nums= [1,4,10,6,5,42,9,8,12]
if search(nums,6):
    print('Found it! 6 is in the list!')
else:
    print('Did not find 6 in the list.')
```

- Example of **linear search**.

Design Pattern: Linear Search

```
def search(nums, locate):
    found = False
    i = 0
    while not found and i < len(nums):
        print(nums[i])
        if locate == nums[i]:
            found = True
        else:
            i = i+1
    return(found)

nums= [1,4,10,6,5,42,9,8,12]
if search(nums,6):
    print('Found it! 6 is in the list!')
else:
    print('Did not find 6 in the list.')
```

- Example of **linear search**.
- Start at the beginning of the list.

Design Pattern: Linear Search

```
def search(nums, locate):
    found = False
    i = 0
    while not found and i < len(nums):
        print(nums[i])
        if locate == nums[i]:
            found = True
        else:
            i = i+1
    return(found)

nums= [1,4,10,6,5,42,9,8,12]
if search(nums,6):
    print('Found it! 6 is in the list!')
else:
    print('Did not find 6 in the list.')
```

- Example of **linear search**.
- Start at the beginning of the list.
- Look at each item, one-by-one.

Design Pattern: Linear Search

```
def search(nums, locate):
    found = False
    i = 0
    while not found and i < len(nums):
        print(nums[i])
        if locate == nums[i]:
            found = True
        else:
            i = i+1
    return(found)

nums= [1,4,10,6,5,42,9,8,12]
if search(nums,6):
    print('Found it! 6 is in the list!')
else:
    print('Did not find 6 in the list.')
```

- Example of **linear search**.
- Start at the beginning of the list.
- Look at each item, one-by-one.
- Stopping, when found, or the end of list is reached.

Today's Topics

- Design Patterns: Searching
- **Python Recap**
- Machine Language
- Machine Language: Jumps & Loops

- Final Exam: Format

Python & Circuits Review: 9 Classes in 10 Minutes

A whirlwind tour of the semester, so far...

Class 1: print(), loops, comments, & turtles

Class 1: print(), loops, comments, & turtles

- Introduced comments & print():

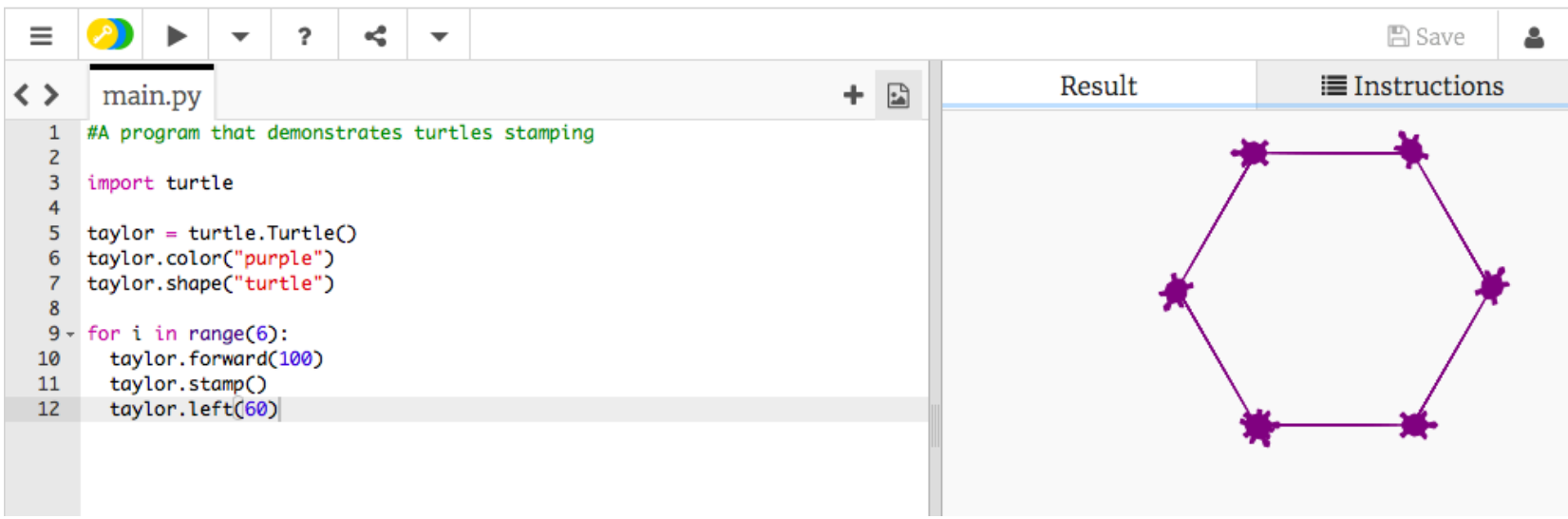
```
#Name:  Thomas Hunter                          ← These lines are comments
#Date:  September 1, 2017                      ← (for us, not computer to read)
#This program prints:  Hello, World!           ← (this one also)

print("Hello, World!")                         ← Prints the string "Hello, World!" to the screen
```

Class 1: print(), loops, comments, & turtles

- Introduced comments & print():

```
#Name:  Thomas Hunter                          ← These lines are comments
#Date:  September 1, 2017                      ← (for us, not computer to read)
#This program prints:  Hello, World!           ← (this one also)

print("Hello, World!")                         ← Prints the string "Hello, World!" to the screen
```

- As well as definite loops & the turtle package:



Class 1: variables, data types, more on loops & range()

Class 1: variables, data types, more on loops & range()

- A **variable** is a reserved memory location for storing a value.

Class 1: variables, data types, more on loops & range()

- A **variable** is a reserved memory location for storing a value.
- Different kinds, or **types**, of values need different amounts of space:
  - **int**: integer or whole numbers

Class 1: variables, data types, more on loops & range()

- A **variable** is a reserved memory location for storing a value.
- Different kinds, or **types**, of values need different amounts of space:
  - **int**: integer or whole numbers
  - **float**: floating point or real numbers

Class 1: variables, data types, more on loops & range()

- A **variable** is a reserved memory location for storing a value.
- Different kinds, or **types**, of values need different amounts of space:
  - **int**: integer or whole numbers
  - **float**: floating point or real numbers
  - **string**: sequence of characters

Class 1: variables, data types, more on loops & range()

- A **variable** is a reserved memory location for storing a value.
- Different kinds, or **types**, of values need different amounts of space:
  - **int**: integer or whole numbers
  - **float**: floating point or real numbers
  - **string**: sequence of characters
  - **list**: a sequence of items

Class 1: variables, data types, more on loops & range()

- A **variable** is a reserved memory location for storing a value.
- Different kinds, or **types**, of values need different amounts of space:
  - **int**: integer or whole numbers
  - **float**: floating point or real numbers
  - **string**: sequence of characters
  - **list**: a sequence of items
    e.g. `[3, 1, 4, 5, 9]` or `['violet','purple','indigo']`

Class 1: variables, data types, more on loops & range()

- A **variable** is a reserved memory location for storing a value.
- Different kinds, or **types**, of values need different amounts of space:
  - **int**: integer or whole numbers
  - **float**: floating point or real numbers
  - **string**: sequence of characters
  - **list**: a sequence of items
    e.g. `[3, 1, 4, 5, 9]` or `['violet','purple','indigo']`
  - **class variables**: for complex objects, like turtles.

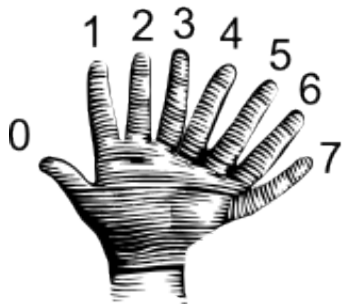
Class 1: variables, data types, more on loops & range()

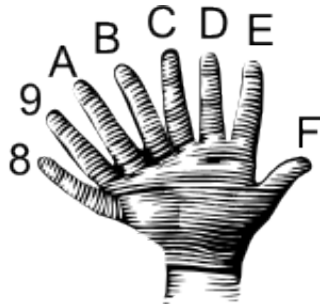
- A **variable** is a reserved memory location for storing a value.
- Different kinds, or **types**, of values need different amounts of space:
  - **int**: integer or whole numbers
  - **float**: floating point or real numbers
  - **string**: sequence of characters
  - **list**: a sequence of items
    e.g. `[3, 1, 4, 5, 9]` or `['violet','purple','indigo']`
  - **class variables**: for complex objects, like turtles.
- More on loops & ranges:

```
1  #Predict what will be printed:
2
3  for num in [2,4,6,8,10]:
4      print(num)
5
6  sum = 0
7  for x in range(0,12,2):
8      print(x)
9      sum = sum + x
10
11 print(sum)
12
13 for c in "ABCD":
14     print(c)
```

Class 2: colors, hex, slices, numpy & images

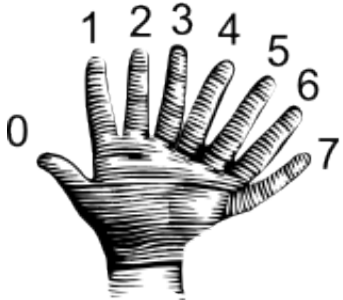
Color Name	HEX	Color
Black	#000000	
Navy	#000080	
DarkBlue	#00008B	
MediumBlue	#0000CD	
Blue	#0000FF	

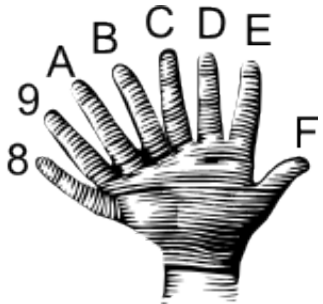


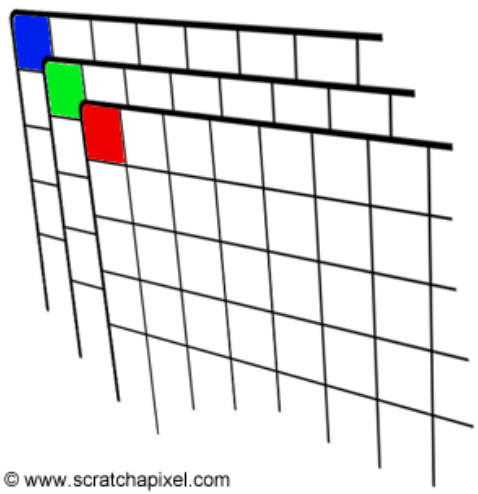


Class 2: colors, hex, slices, numpy & images

Color Name	HEX	Color
Black	#000000	
Navy	#000080	
DarkBlue	#00008B	
MediumBlue	#0000CD	
Blue	#0000FF	

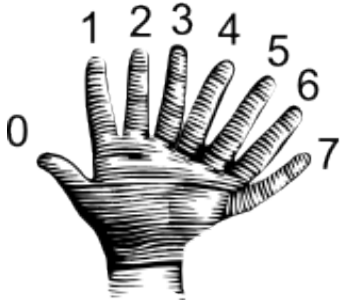


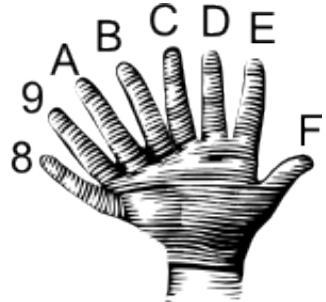


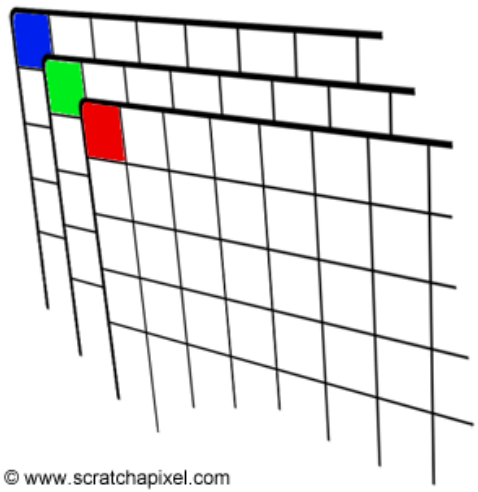


Class 2: colors, hex, slices, numpy & images

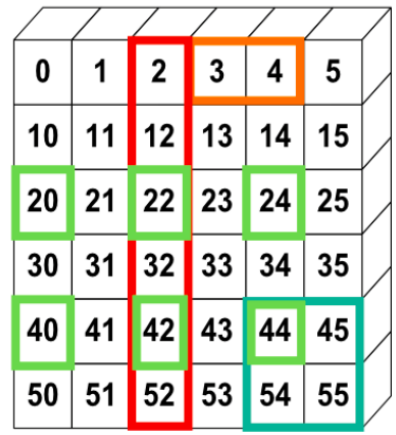
Color Name	HEX	Color
Black	#000000	
Navy	#000080	
DarkBlue	#00008B	
MediumBlue	#0000CD	
Blue	#0000FF	



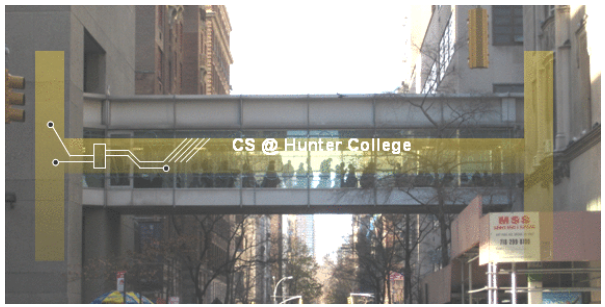




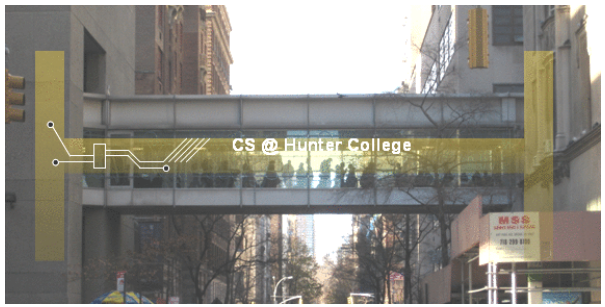

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```

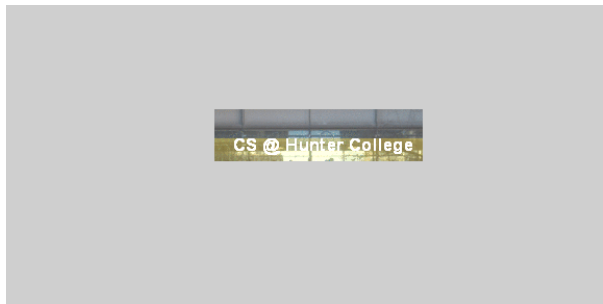


Class 3: design problem (cropping images) & decisions



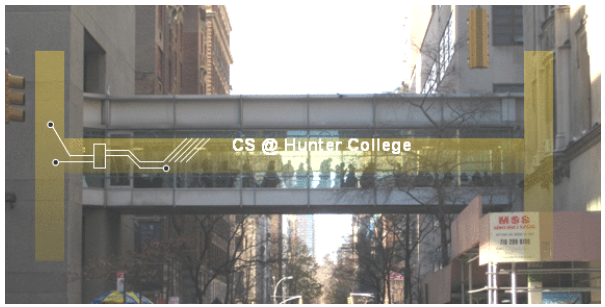
Class 3: design problem (cropping images) & decisions





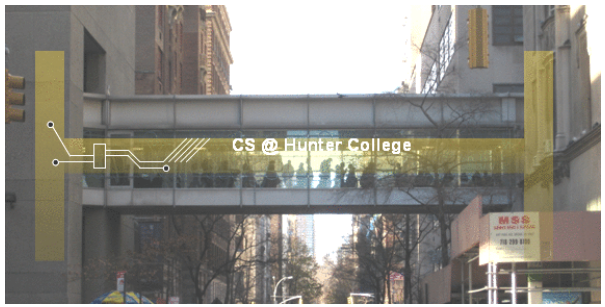
- First: specify inputs/outputs. *Input file name, output file name, upper, lower, left, right ("bounding box")*

Class 3: design problem (cropping images) & decisions



- First: specify inputs/outputs. *Input file name, output file name, upper, lower, left, right ("bounding box")*
- Next: write pseudocode.
  1. Import numpy and pyplot.
  2. Ask user for file names and dimensions for cropping.
  3. Save input file to an array.
  4. Copy the cropped portion to a new array.
  5. Save the new array to the output file.

Class 3: design problem (cropping images) & decisions



- First: specify inputs/outputs. *Input file name, output file name, upper, lower, left, right ("bounding box")*
- Next: write pseudocode.
  1. Import numpy and pyplot.
  2. Ask user for file names and dimensions for cropping.
  3. Save input file to an array.
  4. Copy the cropped portion to a new array.
  5. Save the new array to the output file.
- Next: translate to Python.

Class 3: design problem (cropping images) & decisions

```
yearBorn = int(input('Enter year born: '))
if yearBorn < 1946:
    print("Greatest Generation")
elif yearBorn <= 1964:
    print("Baby Boomer")
elif yearBorn <= 1984:
    print("Generation X")
elif yearBorn <= 2004:
    print("Millennial")
else:
    print("TBD")

x = int(input('Enter number: '))
if x % 2 == 0:
    print('Even number')
else:
    print('Odd number')
```

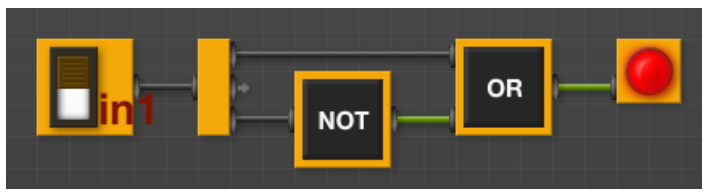
Class 4: logical operators, truth tables & logical circuits

```
origin = "Indian Ocean"
winds = 100
if (winds > 74):
    print("Major storm, called a ", end="")
    if origin == "Indian Ocean" or origin == "South Pacific":
        print("cyclone.")
    elif origin == "North Pacific":
        print("typhoon.")
    else:
        print("hurricane.")

visibility = 0.2
winds = 40
conditions = "blowing snow"
if (winds > 35) and (visibility < 0.25) and \
      (conditions == "blowing snow" or conditions == "heavy snow"):
    print("Blizzard!")
```

Class 4: logical operators, truth tables & logical circuits

```
origin = "Indian Ocean"
winds = 100
if (winds > 74):
    print("Major storm, called a ", end="")
    if origin == "Indian Ocean" or origin == "South Pacific":
        print("cyclone.")
    elif origin == "North Pacific":
        print("typhoon.")
    else:
        print("hurricane.")

visibility = 0.2
winds = 40
conditions = "blowing snow"
if (winds > 35) and (visibility < 0.25) and \
      (conditions == "blowing snow" or conditions == "heavy snow"):
    print("Blizzard!")
```

in1		in2	*returns:*
False	and	False	False
False	and	True	False
True	and	False	False
True	and	True	True



Class 5: structured data, pandas, & more design

```
Source:  https://en.wikipedia.org/wiki/Demographics_of_New_York_City,,,,,,
All population figures are consistent with present-day boundaries.,,,,,,
First census after the consolidation of the five boroughs,,,,,,
,,,,,,
,,,,,,
Year,Manhattan,Brooklyn,Queens,Bronx,Staten Island,Total
1698,4937,2017,,,727,7681
1771,21863,3623,,,2847,28423
1790,33131,4549,6159,1781,3827,49447
1800,60515,5740,6642,1755,4563,79215
1810,96373,8303,7444,2267,5347,119734
1820,123706,11187,8246,2782,6135,152056
1830,202589,20535,9049,3023,7082,242278
1840,312710,47613,14480,5346,10965,391114
1850,515547,138882,18593,8032,15061,696115
1860,813669,279122,32903,23593,25492,1174779
1870,942292,419921,45468,37393,33029,1478103
1880,1164673,599495,56559,51980,38991,1911698
1890,1441216,838547,87050,88908,51693,2507414
1900,1850093,1166582,152999,200507,67021,3437202
1910,2331542,1634351,284041,430980,85969,4766883
1920,2284103,2018356,469042,732016,116531,5620048
1930,1867312,2560401,1079129,1265258,158346,6930446
1940,1889924,2698285,1297634,1394711,174441,7454995
1950,1960101,2738175,1550849,1451277,191555,7891957
1960,1698281,2627319,1809578,1424815,221991,7781984
1970,1539233,2602012,1986473,1471701,295443,7894862
1980,1428285,2230936,1891325,1168972,352121,7071639
1990,1487536,2300664,1951598,1203789,378977,7322564
2000,1537195,2465326,2229379,1332650,443728,8008278
2010,1585873,2504700,2230722,1385108,468730,8175133
2015,1644518,2636735,2339150,1455444,474558,8550405
```

`nycHistPop.csv`

In Lab 6

Class 5: structured data, pandas, & more design

```
import matplotlib.pyplot as plt
import pandas as pd
```

```
Source:  https://en.wikipedia.org/wiki/Demographics_of_New_York_City,,,,,,
All population figures are consistent with present-day boundaries.,,,,,,
First census after the consolidation of the five boroughs,,,,,,
,,,,,,
,,,,,,
Year,Manhattan,Brooklyn,Queens,Bronx,Staten Island,Total
1698,4937,2017,,,727,7681
1771,21863,3623,,,2847,28423
1790,33131,4549,6159,1781,3827,49447
1800,60515,5740,6642,1755,4563,79215
1810,96373,8303,7444,2267,5347,119734
1820,123706,11187,8246,2782,6135,152056
1830,202589,20535,9049,3023,7082,242278
1840,312710,47613,14480,5346,10965,391114
1850,515547,138882,18593,8032,15061,696115
1860,813669,279122,32903,23593,25492,1174779
1870,942292,419921,45468,37393,33029,1478103
1880,1164673,599495,56559,51980,38991,1911698
1890,1441216,838547,87050,88908,51693,2507414
1900,1850093,1166582,152999,200507,67021,3437202
1910,2331542,1634351,284041,430980,85969,4766883
1920,2284103,2018356,469042,732016,116531,5620048
1930,1867312,2560401,1079129,1265258,158346,6930446
1940,1889924,2698285,1297634,1394711,174441,7454995
1950,1960101,2738175,1550849,1451277,191555,7891957
1960,1698281,2627319,1809578,1424815,221991,7781984
1970,1539233,2602012,1986473,1471701,295443,7894862
1980,1428285,2230936,1891325,1168972,352121,7071639
1990,1487536,2300664,1951598,1203789,378977,7322564
2000,1537195,2465326,2229379,1332650,443728,8008278
2010,1585873,2504700,2230722,1385108,468730,8175133
2015,1644518,2636735,2339150,1455444,474558,8550405
```

`nycHistPop.csv`

In Lab 6

Class 5: structured data, pandas, & more design

```
import matplotlib.pyplot as plt
import pandas as pd

pop = pd.read_csv('nycHistPop.csv',skiprows=5)
```

```
Source:  https://en.wikipedia.org/wiki/Demographics_of_New_York_City,,,,,,
All population figures are consistent with present-day boundaries.,,,,,,
First census after the consolidation of the five boroughs,,,,,,
,,,,,,
,,,,,,
Year,Manhattan,Brooklyn,Queens,Bronx,Staten Island,Total
1698,4937,2017,,,727,7681
1771,21863,3623,,,2847,28423
1790,33131,4549,6159,1781,3827,49447
1800,60515,5740,6642,1755,4563,79215
1810,96373,8303,7444,2267,5347,119734
1820,123706,11187,8246,2782,6135,152056
1830,202589,20535,9049,3023,7082,242278
1840,312710,47613,14480,5346,10965,391114
1850,515547,138882,18593,8032,15061,696115
1860,813669,279122,32903,23593,25492,1174779
1870,942292,419921,45468,37393,33029,1478103
1880,1164673,599495,56559,51980,38991,1911698
1890,1441216,838547,87050,88908,51693,2507414
1900,1850093,1166582,152999,200507,67021,3437202
1910,2331542,1634351,284041,430980,85969,4766883
1920,2284103,2018356,469042,732016,116531,5620048
1930,1867312,2560401,1079129,1265258,158346,6930446
1940,1889924,2698285,1297634,1394711,174441,7454995
1950,1960101,2738175,1550849,1451277,191555,7891957
1960,1698281,2627319,1809578,1424815,221991,7781984
1970,1539233,2602012,1986473,1471701,295443,7894862
1980,1428285,2230936,1891325,1168972,352121,7071639
1990,1487536,2300664,1951598,1203789,378977,7322564
2000,1537195,2465326,2229379,1332650,443728,8008278
2010,1585873,2504700,2230722,1385108,468730,8175133
2015,1644518,2636735,2339150,1455444,474558,8550405
```

nycHistPop.csv

In Lab 6

Class 5: structured data, pandas, & more design

```
import matplotlib.pyplot as plt
import pandas as pd

pop = pd.read_csv('nycHistPop.csv',skiprows=5)

pop.plot(x="Year")
plt.show()
```

```
Source:  https://en.wikipedia.org/wiki/Demographics_of_New_York_City,,,,,,
All population figures are consistent with present-day boundaries.,,,,,,
First census after the consolidation of the five boroughs,,,,,,
,,,,,,
,,,,,,
Year,Manhattan,Brooklyn,Queens,Bronx,Staten Island,Total
1698,4937,2017,,,727,7681
1771,21863,3623,,,2847,28423
1790,33131,4549,6159,1781,3827,49447
1800,60515,5740,6642,1755,4563,79215
1810,96373,8303,7444,2267,5347,119734
1820,123706,11187,8246,2782,6135,152056
1830,202589,20535,9049,3023,7082,242278
1840,312710,47613,14480,5346,10965,391114
1850,515547,138882,18593,8032,15061,696115
1860,813669,279122,32903,23593,25492,1174779
1870,942292,419921,45468,37393,33029,1478103
1880,1164673,599495,56559,51980,38991,1911698
1890,1441216,838547,87050,88908,51693,2507414
1900,1850093,1166582,152999,200507,67021,3437202
1910,2331542,1634351,284041,430980,85969,4766883
1920,2284103,2018356,469042,732016,116531,5620048
1930,1867312,2560401,1079129,1265258,158346,6930446
1940,1889924,2698285,1297634,1394711,174441,7454995
1950,1960101,2738175,1550849,1451277,191555,7891957
1960,1698281,2627319,1809578,1424815,221991,7781984
1970,1539233,2602012,1986473,1471701,295443,7894862
1980,1428285,2230936,1891325,1168972,352121,7071639
1990,1487536,2300664,1951598,1203789,378977,7322564
2000,1537195,2465326,2229379,1332650,443728,8008278
2010,1585873,2504700,2230722,1385108,468730,8175133
2015,1644518,2636735,2339150,1455444,474558,8550405
```

nycHistPop.csv

In Lab 6

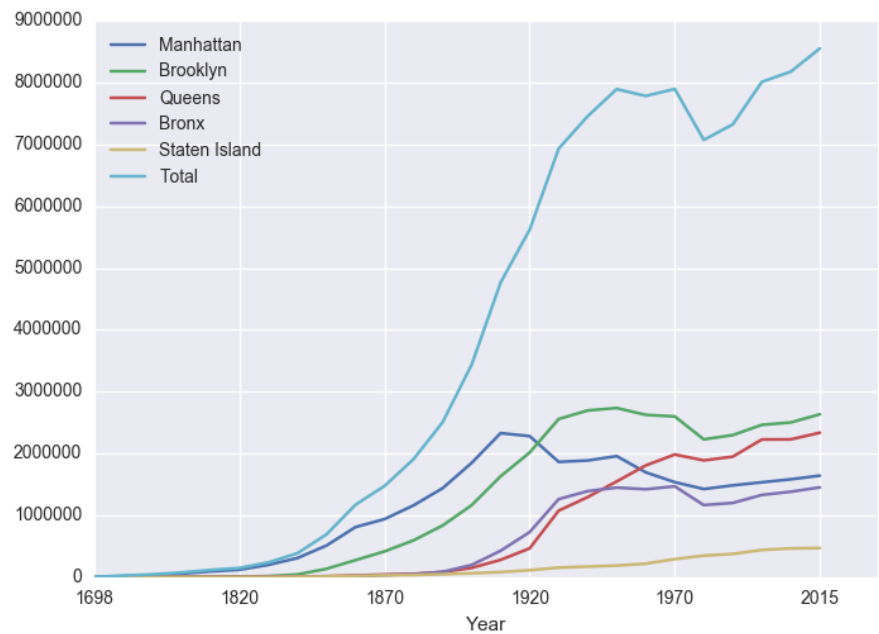
Class 5: structured data, pandas, & more design

```
import matplotlib.pyplot as plt
import pandas as pd

pop = pd.read_csv('nycHistPop.csv',skiprows=5)

pop.plot(x="Year")
plt.show()
```

```
Source:  https://en.wikipedia.org/wiki/Demographics_of_New_York_City,,,,,,
All population figures are consistent with present-day boundaries.,,,,,,
First census after the consolidation of the five boroughs,,,,,,
,,,,,,
,,,,,,
Year,Manhattan,Brooklyn,Queens,Bronx,Staten Island,Total
1698,4937,2017,,,727,7681
1771,21863,3623,,,2847,28423
1790,33131,4549,6159,1781,3827,49447
1800,60515,5740,6642,1755,4563,79215
1810,96373,8303,7444,2267,5347,119734
1820,123706,11187,8246,2782,6135,152056
1830,202589,20535,9049,3023,7082,242278
1840,312710,47613,14480,5346,10965,391114
1850,515547,138882,18593,8032,15061,696115
1860,813669,279122,32903,23593,25492,1174779
1870,942292,419921,45468,37393,33029,1478103
1880,1164673,599495,56559,51980,38991,1911698
1890,1441216,838547,87050,88908,51693,2507414
1900,1850093,1166582,152999,200507,67021,3437202
1910,2331542,1634351,284041,430980,85969,4766883
1920,2284103,2018356,469042,732016,116531,5620048
1930,1867312,2560401,1079129,1265258,158346,6930446
1940,1889924,2698285,1297634,1394711,174441,7454995
1950,1960101,2738175,1550849,1451277,191555,7891957
1960,1698281,2627319,1809578,1424815,221991,7781984
1970,1539233,2602012,1986473,1471701,295443,7894862
1980,1428285,2230936,1891325,1168972,352121,7071639
1990,1487536,2300664,1951598,1203789,378977,7322564
2000,1537195,2465326,2229379,1332650,443728,8008278
2010,1585873,2504700,2230722,1385108,468730,8175133
2015,1644518,2636735,2339150,1455444,474558,8550405
```

`nycHistPop.csv`

In Lab 6



Class 6: functions

```
#Name:  your name here
#Date:  October 2017
#This program, uses functions,
#     says hello to the world!

def main():
     print("Hello, World!")

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

- Functions are a way to break code into pieces, that can be easily reused.

Class 6: functions

```
#Name:  your name here
#Date:  October 2017
#This program, uses functions,
#     says hello to the world!

def main():
     print("Hello, World!")

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

- Functions are a way to break code into pieces, that can be easily reused.
- Many languages require that all code must be organized with functions.

Class 6: functions

```
#Name:  your name here
#Date:  October 2017
#This program, uses functions,
#     says hello to the world!

def main():
     print("Hello, World!")

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

- Functions are a way to break code into pieces, that can be easily reused.
- Many languages require that all code must be organized with functions.
- The opening function is often called `main()`

Class 6: functions

```
#Name:  your name here
#Date:  October 2017
#This program, uses functions,
#     says hello to the world!

def main():
     print("Hello, World!")

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

- Functions are a way to break code into pieces, that can be easily reused.
- Many languages require that all code must be organized with functions.
- The opening function is often called `main()`
- You **call** or **invoke** a function by typing its name, followed by any inputs, surrounded by parenthesis:

Class 6: functions

```
#Name:  your name here
#Date:  October 2017
#This program, uses functions,
#     says hello to the world!

def main():
     print("Hello, World!")

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

- Functions are a way to break code into pieces, that can be easily reused.
- Many languages require that all code must be organized with functions.
- The opening function is often called `main()`
- You **call** or **invoke** a function by typing its name, followed by any inputs, surrounded by parenthesis: Example: `print("Hello", "World")`

Class 6: functions

```
#Name:  your name here
#Date:  October 2017
#This program, uses functions,
#     says hello to the world!

def main():
     print("Hello, World!")

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

- Functions are a way to break code into pieces, that can be easily reused.
- Many languages require that all code must be organized with functions.
- The opening function is often called `main()`
- You **call** or **invoke** a function by typing its name, followed by any inputs, surrounded by parenthesis: Example: `print("Hello", "World")`
- Can write, or **define** your own functions,

Class 6: functions

```
#Name:  your name here
#Date:  October 2017
#This program, uses functions,
#     says hello to the world!

def main():
     print("Hello, World!")

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

- Functions are a way to break code into pieces, that can be easily reused.
- Many languages require that all code must be organized with functions.
- The opening function is often called `main()`
- You **call** or **invoke** a function by typing its name, followed by any inputs, surrounded by parenthesis: Example: `print("Hello", "World")`
- Can write, or **define** your own functions, which are stored, until invoked or called.

Class 7: function parameters, github

- Functions can have **input parameters**.

```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)

lunch = float(input('Enter lunch total: '))
lTip = float(input('Enter lunch tip:' ))
lTotal = totalWithTax(lunch, lTip)
print('Lunch total is', lTotal)

dinner= float(input('Enter dinner total: '))
dTip = float(input('Enter dinner tip:' ))
dTotal = totalWithTax(dinner, dTip)
print('Dinner total is', dTotal)
```

Class 7: function parameters, github

```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)

lunch = float(input('Enter lunch total: '))
lTip = float(input('Enter lunch tip:' ))
lTotal = totalWithTax(lunch, lTip)
print('Lunch total is', lTotal)

dinner= float(input('Enter dinner total: '))
dTip = float(input('Enter dinner tip:' ))
dTotal = totalWithTax(dinner, dTip)
print('Dinner total is', dTotal)
```

- Functions can have **input parameters**.
- Surrounded by parenthesis, both in the function definition, and in the function call (invocation).

Class 7: function parameters, github

```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)

lunch = float(input('Enter lunch total: '))
lTip = float(input('Enter lunch tip:' ))
lTotal = totalWithTax(lunch, lTip)
print('Lunch total is', lTotal)

dinner= float(input('Enter dinner total: '))
dTip = float(input('Enter dinner tip:' ))
dTotal = totalWithTax(dinner, dTip)
print('Dinner total is', dTotal)
```

- Functions can have **input parameters**.
- Surrounded by parenthesis, both in the function definition, and in the function call (invocation).
- The "placeholders" in the function definition: **formal parameters**.

Class 7: function parameters, github

```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)

lunch = float(input('Enter lunch total: '))
lTip = float(input('Enter lunch tip:' ))
lTotal = totalWithTax(lunch, lTip)
print('Lunch total is', lTotal)

dinner= float(input('Enter dinner total: '))
dTip = float(input('Enter dinner tip:' ))
dTotal = totalWithTax(dinner, dTip)
print('Dinner total is', dTotal)
```

- Functions can have **input parameters**.
- Surrounded by parenthesis, both in the function definition, and in the function call (invocation).
- The "placeholders" in the function definition: **formal parameters**.
- The ones in the function call: **actual parameters**

Class 7: function parameters, github

```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)

lunch = float(input('Enter lunch total: '))
lTip = float(input('Enter lunch tip:' ))
lTotal = totalWithTax(lunch, lTip)
print('Lunch total is', lTotal)

dinner= float(input('Enter dinner total: '))
dTip = float(input('Enter dinner tip:' ))
dTotal = totalWithTax(dinner, dTip)
print('Dinner total is', dTotal)
```

- Functions can have **input parameters**.
- Surrounded by parenthesis, both in the function definition, and in the function call (invocation).
- The "placeholders" in the function definition: **formal parameters**.
- The ones in the function call: **actual parameters**
- Functions can also **return values** to where it was called.

Class 7: function parameters, github

```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)

lunch = float(input('Enter lunch total: '))
lTip = float(input('Enter lunch tip:' ))
lTotal = totalWithTax(lunch, lTip)
print('Lunch total is', lTotal)

dinner= float(input('Enter dinner total: '))
dTip = float(input('Enter dinner tip:' ))
dTotal = totalWithTax(dinner, dTip)
print('Dinner total is', dTotal)
```

Formal Parameters

Actual Parameters

- Functions can have **input parameters**.
- Surrounded by parenthesis, both in the function definition, and in the function call (invocation).
- The "placeholders" in the function definition: **formal parameters**.
- The ones in the function call: **actual parameters**.
- Functions can also **return values** to where it was called.

Class 8: top-down design, folium, loops, and random()


```
def main():
    dataF = getData()
    latColName, lonColName = getColumnNames()
    lat, lon = getLocale()
    cityMap = folium.Map(location = [lat,lon], tiles = 'cartodbpositron',zoom_start=11)
    dotAllPoints(cityMap,dataF,latColName,lonColName)
    markAndFindClosest(cityMap,dataF,latColName,lonColName,lat,lon)
    writeMap(cityMap)
```

Class 9: more on loops, max design pattern, random()

```
dist = int(input('Enter distance: '))
while dist < 0:
    print('Distances cannot be negative.')
    dist = int(input('Enter distance: '))

print('The distance entered is', dist)
```

- Indefinite (while) loops allow you to repeat a block of code as long as a condition holds.

```
import turtle
import random

trey = turtle.Turtle()
trey.speed(10)

for i in range(100):
  trey.forward(10)
  a = random.randrange(0,360,90)
  trey.right(a)
```

Class 9: more on loops, max design pattern, random()

```
dist = int(input('Enter distance: '))
while dist < 0:
    print('Distances cannot be negative.')
    dist = int(input('Enter distance: '))

print('The distance entered is', dist)
```

- Indefinite (while) loops allow you to repeat a block of code as long as a condition holds.
- Very useful for checking user input for correctness.

```
import turtle
import random

trey = turtle.Turtle()
trey.speed(10)

for i in range(100):
  trey.forward(10)
  a = random.randrange(0,360,90)
  trey.right(a)
```

Class 9: more on loops, max design pattern, random()

```
dist = int(input('Enter distance: '))
while dist < 0:
    print('Distances cannot be negative.')
    dist = int(input('Enter distance: '))

print('The distance entered is', dist)
```

- Indefinite (while) loops allow you to repeat a block of code as long as a condition holds.
- Very useful for checking user input for correctness.
- Python's built-in random package has useful methods for generating random whole numbers and real numbers.

```
import turtle
import random

trey = turtle.Turtle()
trey.speed(10)

for i in range(100):
  trey.forward(10)
  a = random.randrange(0,360,90)
  trey.right(a)
```

Class 9: more on loops, max design pattern, random()

```
dist = int(input('Enter distance: '))
while dist < 0:
    print('Distances cannot be negative.')
    dist = int(input('Enter distance: '))

print('The distance entered is', dist)
```

```
import turtle
import random

trey = turtle.Turtle()
trey.speed(10)

for i in range(100):
  trey.forward(10)
  a = random.randrange(0,360,90)
  trey.right(a)
```

- Indefinite (while) loops allow you to repeat a block of code as long as a condition holds.
- Very useful for checking user input for correctness.
- Python's built-in random package has useful methods for generating random whole numbers and real numbers.
- To use, must include: `import random`.

Class 9: more on loops, max design pattern, random()

```
dist = int(input('Enter distance: '))
while dist < 0:
    print('Distances cannot be negative.')
    dist = int(input('Enter distance: '))

print('The distance entered is', dist)
```

```
import turtle
import random

trey = turtle.Turtle()
trey.speed(10)

for i in range(100):
  trey.forward(10)
  a = random.randrange(0,360,90)
  trey.right(a)
```

- Indefinite (while) loops allow you to repeat a block of code as long as a condition holds.
- Very useful for checking user input for correctness.
- Python's built-in random package has useful methods for generating random whole numbers and real numbers.
- To use, must include: `import random`.
- The max design pattern provides a template for finding maximum value from a list.

Python & Circuits Review: 9 Classes in 10 Minutes

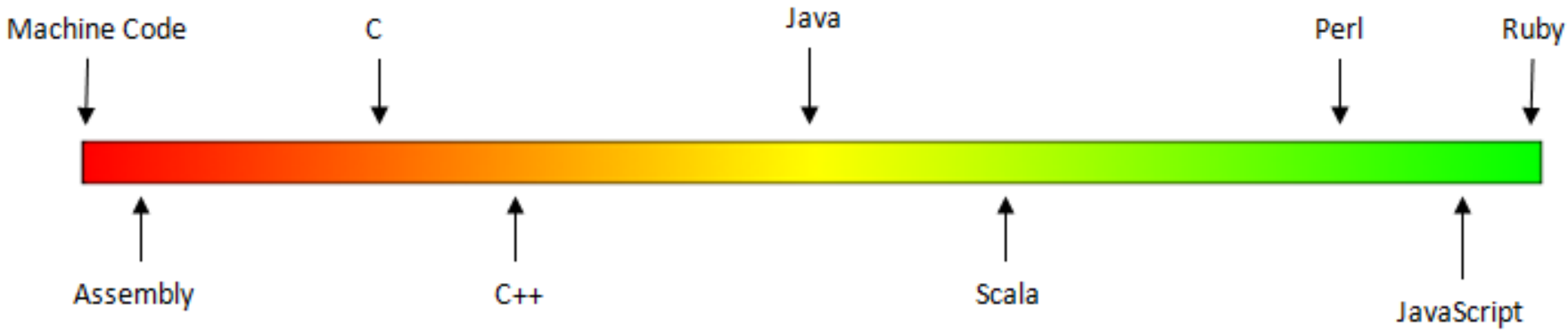
- Input/Output (I/O): `input()` and `print()`; pandas for CSV files
- Types:
  - Primitive: int, float, bool, string;
  - Container: lists (but not dictionaries/hashes or tuples)
- Objects: turtles (used but did not design our own)
- Loops: definite & indefinite
- Conditionals: if-elif-else
- Logical Expressions & Circuits
- Functions: parameters & returns
- Packages:
  - Built-in: turtle, math, random
  - Popular: numpy, matplotlib, pandas, folium

Today's Topics

- Design Patterns: Searching
- Python Recap
- **Machine Language**
- Machine Language: Jumps & Loops

- Final Exam: Format

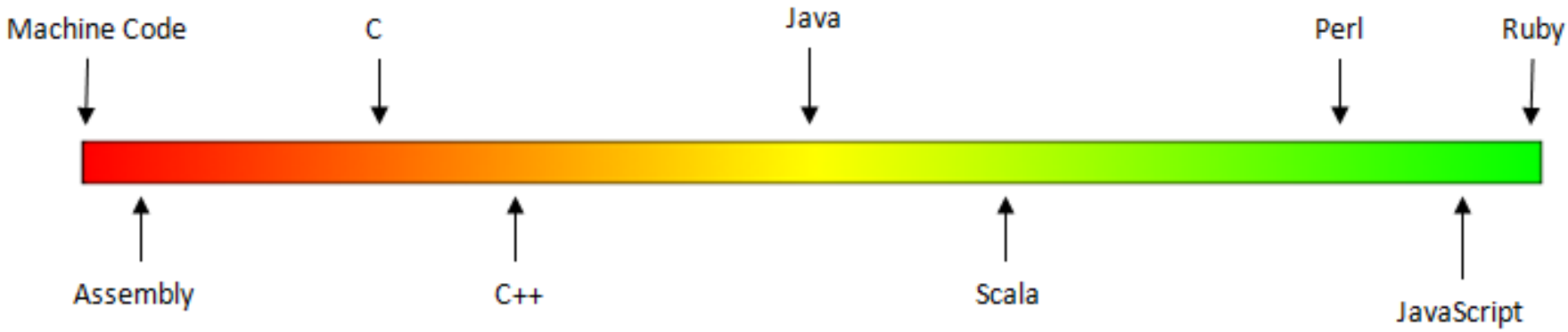
Low-Level vs. High-Level Languages



(codeCommit)

- Can view programming languages on a continuum.

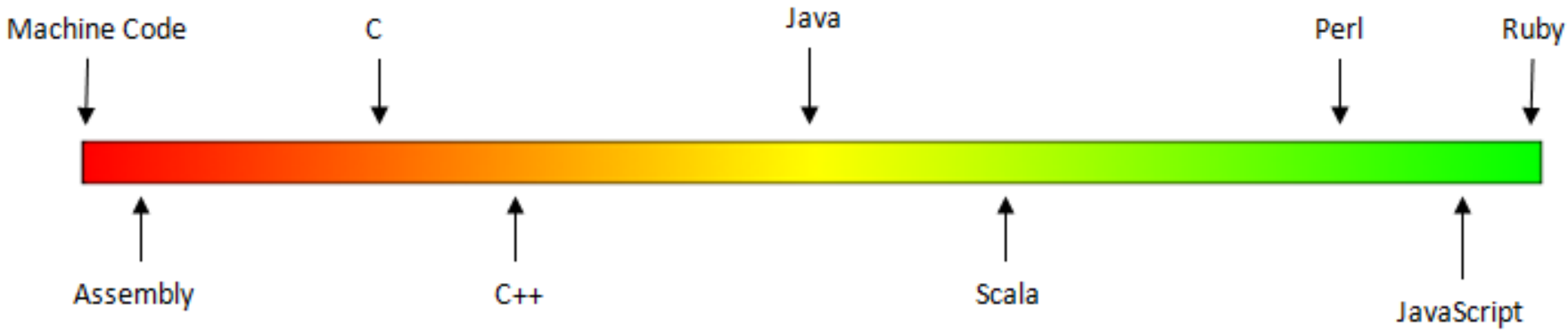
Low-Level vs. High-Level Languages



(codeCommit)

- Can view programming languages on a continuum.
- Those that directly access machine instructions & memory and have little abstraction are **low-level languages**

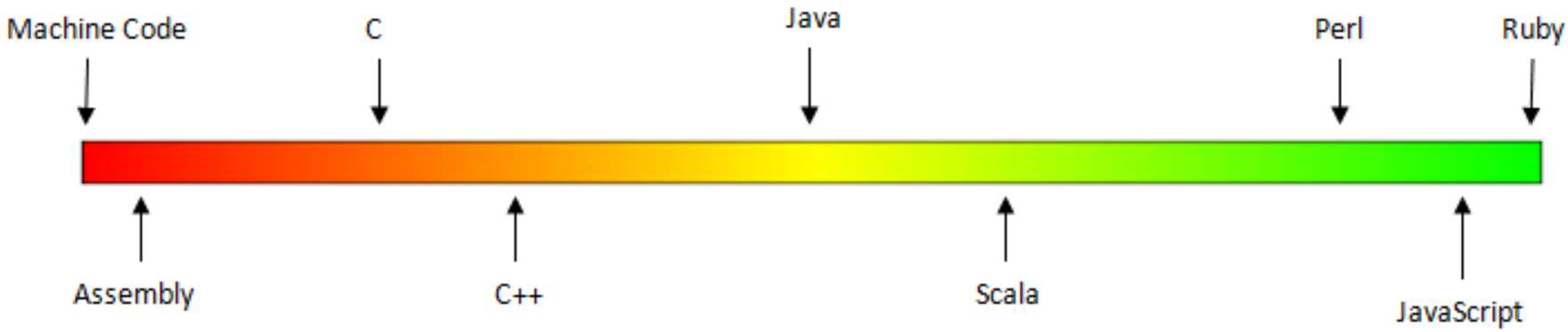
Low-Level vs. High-Level Languages



(codeCommit)

- Can view programming languages on a continuum.
- Those that directly access machine instructions & memory and have little abstraction are **low-level languages**
  (e.g. machine language, assembly language).

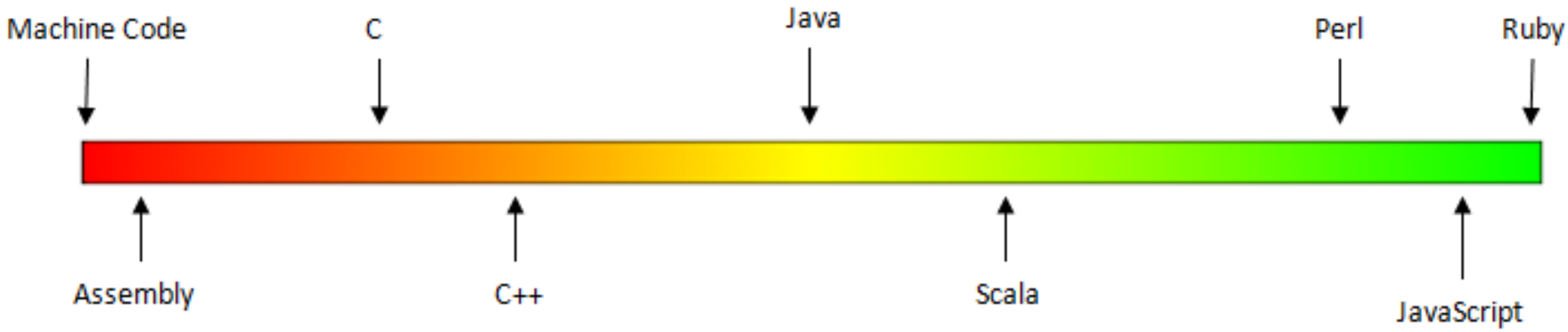
Low-Level vs. High-Level Languages



(codeCommit)

- Can view programming languages on a continuum.
- Those that directly access machine instructions & memory and have little abstraction are **low-level languages** (e.g. machine language, assembly language).
- Those that have strong abstraction (allow programming paradigms independent of the machine details, such as complex variables, functions and looping that do not translate directly into machine code) are called **high-level languages**.

Low-Level vs. High-Level Languages

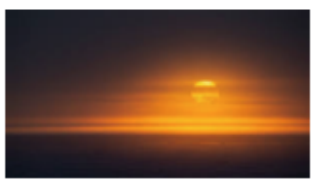


(codeCommit)

- Can view programming languages on a continuum.
- Those that directly access machine instructions & memory and have little abstraction are **low-level languages** (e.g. machine language, assembly language).
- Those that have strong abstraction (allow programming paradigms independent of the machine details, such as complex variables, functions and looping that do not translate directly into machine code) are called **high-level languages**.
- Some languages, like C, are in between– allowing both low level access and high level data structures.

Processing

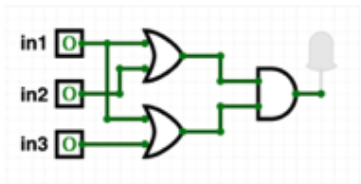
Dies ist ein Blindtext. An ihm lässt sich vieles über die Schrift ablesen, in der er gesetzt ist. Auf den ersten Blick wird der Grauwert der Schriftfläche sichtbar. Dann kann man prüfen, wie gut die Schrift zu lesen ist und wie sie auf den Leser wirkt. Dies ist ein Blindtext. An ihm lässt sich



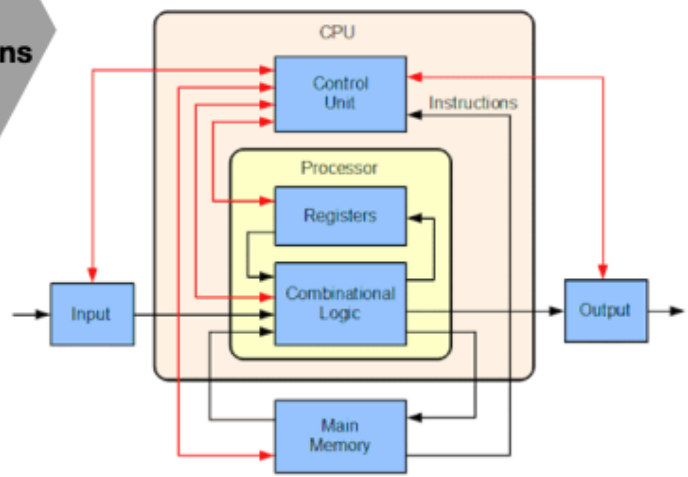
```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)
```

Data & Instructions

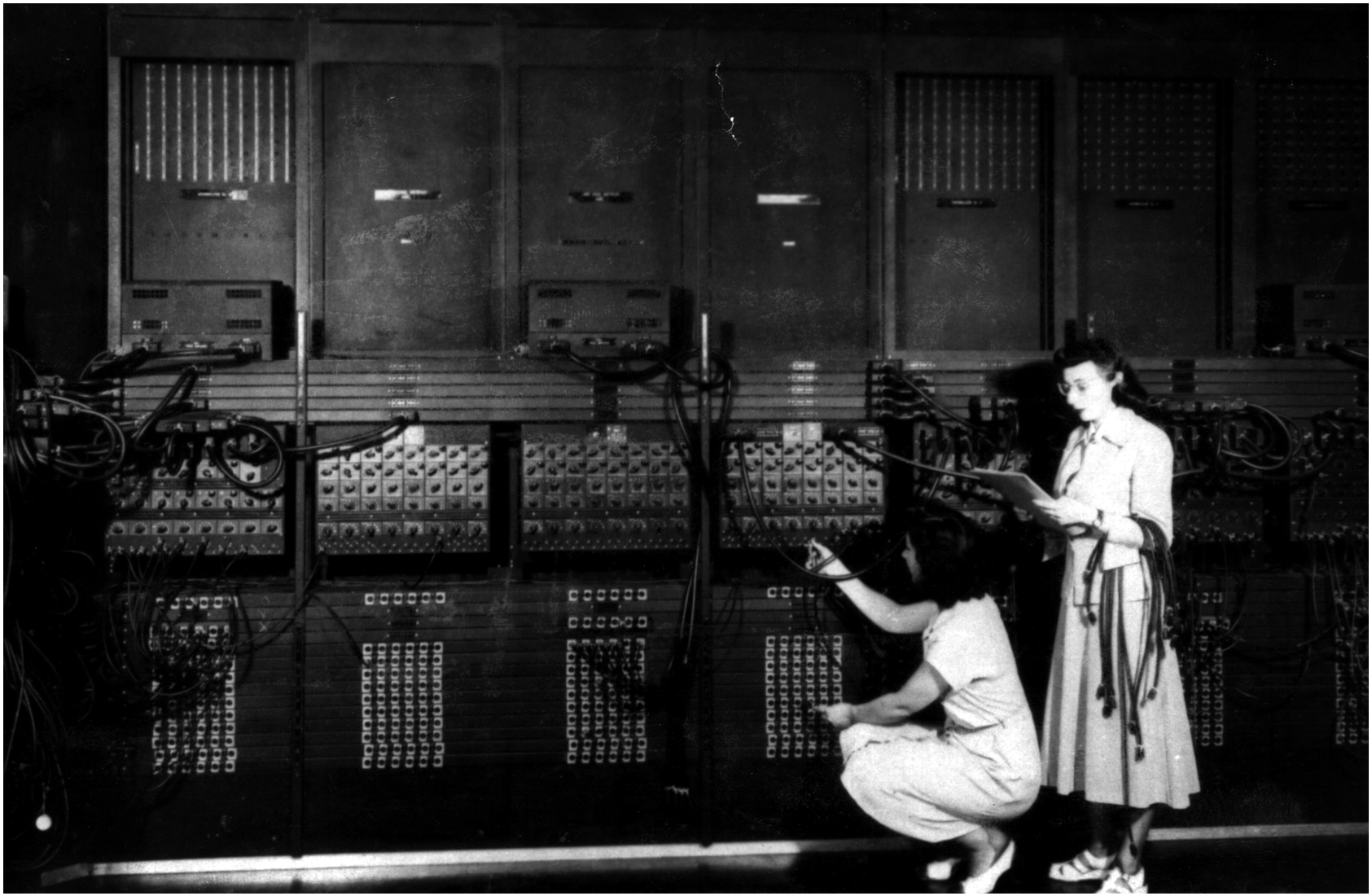
Data & Instructions



Circuits (switches)
On/Off 1/0 Logic
Billions of switches/bits

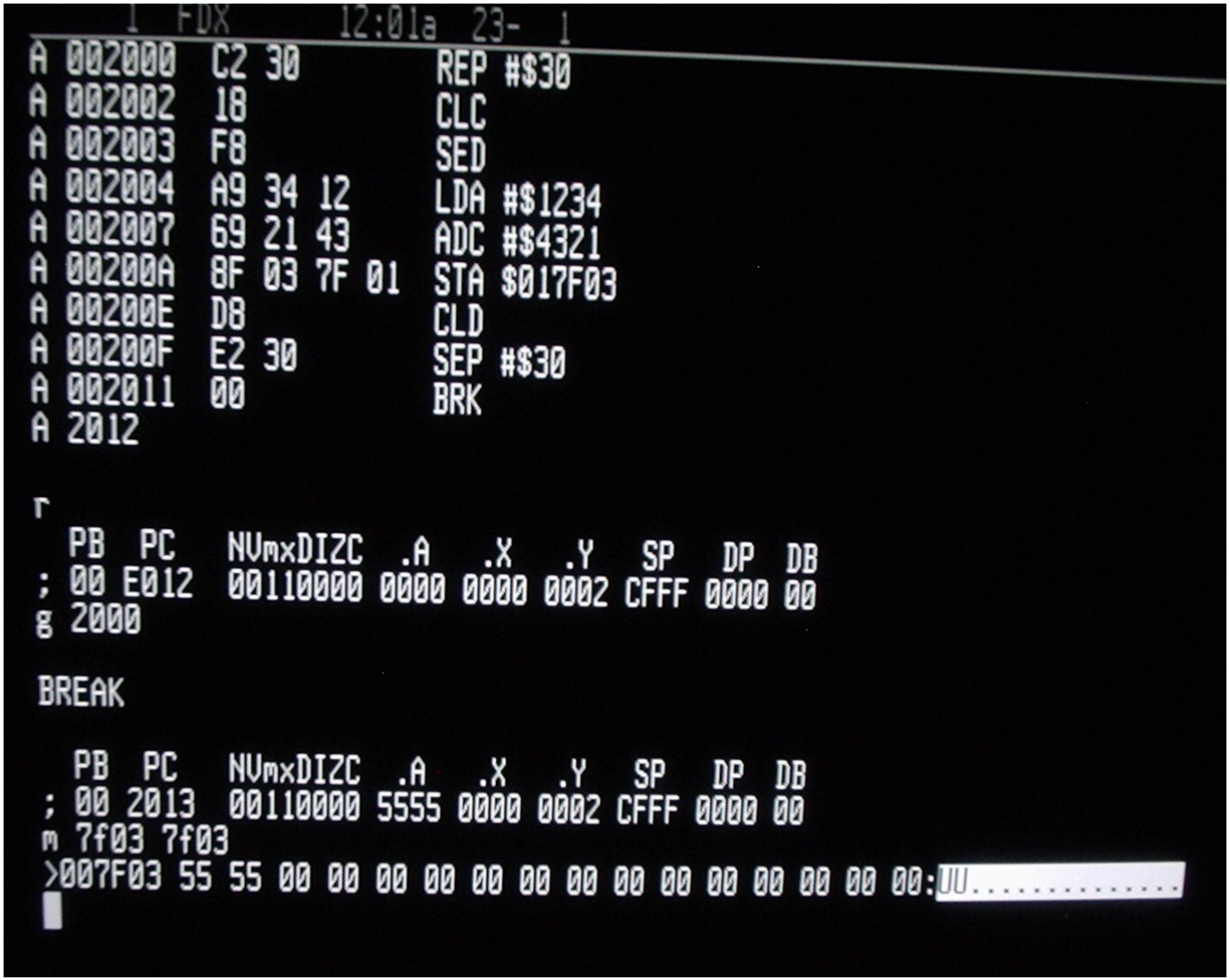


Machine Language



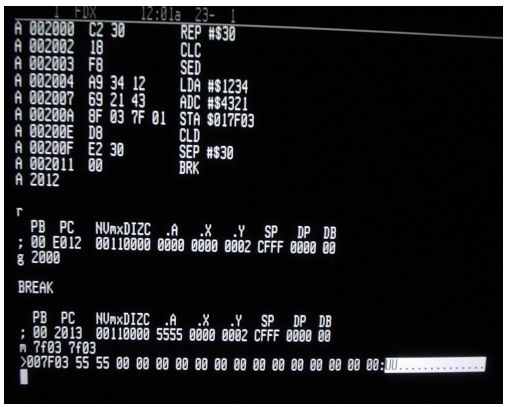
(Ruth Gordon & Ester Gerston programming the ENIAC, UPenn)

Machine Language



(wiki)

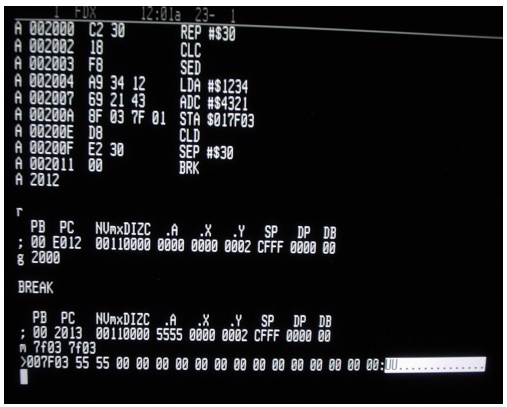
Machine Language



(wiki)

- We will be writing programs in a simplified machine language, WeMIPS.

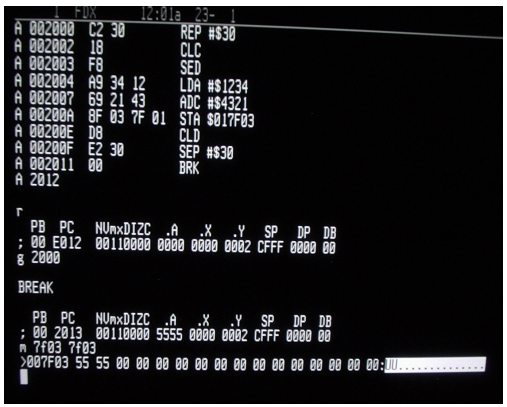
Machine Language



(wiki)

- We will be writing programs in a simplified machine language, WeMIPS.
- It is based on a reduced instruction set computer (RISC) design, originally developed by the MIPS Computer Systems.

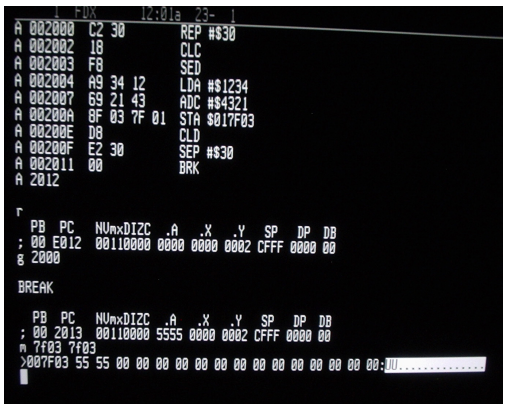
Machine Language



(wiki)

- We will be writing programs in a simplified machine language, WeMIPS.
- It is based on a reduced instruction set computer (RISC) design, originally developed by the MIPS Computer Systems.
- Due to its small set of commands, processors can be designed to run those commands very efficiently.

Machine Language



(wiki)

- We will be writing programs in a simplified machine language, WeMIPS.
- It is based on a reduced instruction set computer (RISC) design, originally developed by the MIPS Computer Systems.
- Due to its small set of commands, processors can be designed to run those commands very efficiently.
- More in future architecture classes....

“Hello World!” in Simplified Machine Language

Line: 3 Go!

Show/Hide Demos

User Guide | Unit Tests | Docs

Addition Doubler | Stav | Looper | Stack Test | Hello World

Code Gen Save String | Interactive | Binary2 Decimal | Decimal2 Binary

Debug

```
1  # Store 'Hello world!' at the top of the stack
2  ADDI $sp, $sp, -13
3  ADDI $t0, $zero, 72 # H
4  SB $t0, 0($sp)
5  ADDI $t0, $zero, 101 # e
6  SB $t0, 1($sp)
7  ADDI $t0, $zero, 108 # l
8  SB $t0, 2($sp)
9  ADDI $t0, $zero, 108 # l
10 SB $t0, 3($sp)
11 ADDI $t0, $zero, 111 # o
12 SB $t0, 4($sp)
13 ADDI $t0, $zero, 32 # (space)
14 SB $t0, 5($sp)
15 ADDI $t0, $zero, 119 # w
16 SB $t0, 6($sp)
17 ADDI $t0, $zero, 111 # o
18 SB $t0, 7($sp)
19 ADDI $t0, $zero, 114 # r
20 SB $t0, 8($sp)
21 ADDI $t0, $zero, 108 # l
22 SB $t0, 9($sp)
23 ADDI $t0, $zero, 100 # d
24 SB $t0, 10($sp)
25 ADDI $t0, $zero, 33 # !
26 SB $t0, 11($sp)
27 ADDI $t0, $zero, 0 # (null)
28 SB $t0, 12($sp)
29
30 ADDI $v0, $zero, 4 # 4 is for print string
31 ADDI $a0, $sp, 0
32 syscall             # print to the log
```

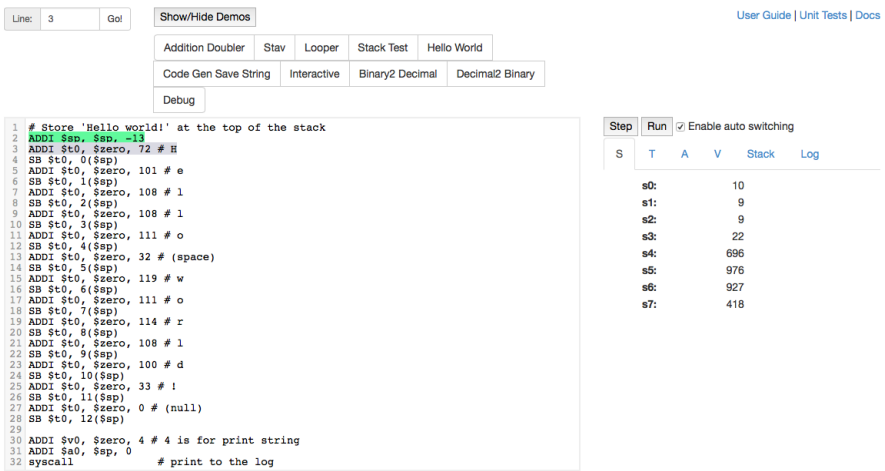
Step | Run | ☑ Enable auto switching

S | T | A | V | Stack | Log

s0:	10
s1:	9
s2:	9
s3:	22
s4:	696
s5:	976
s6:	927
s7:	418

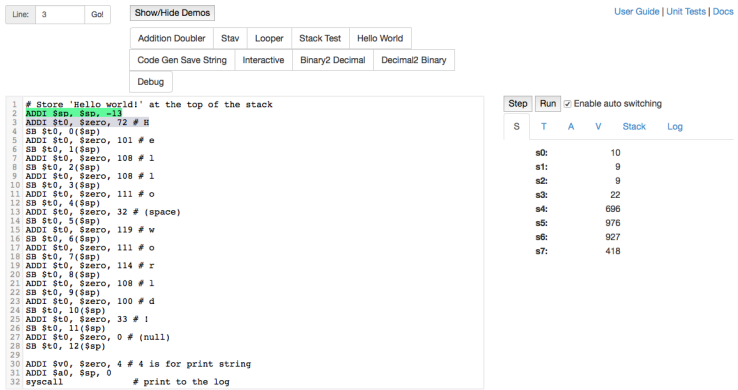
(WeMIPS)

WeMIPS



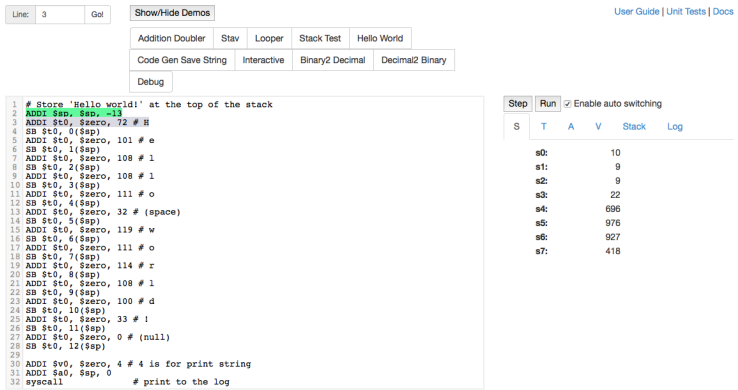
(Demo with WeMIPS)

MIPS Commands



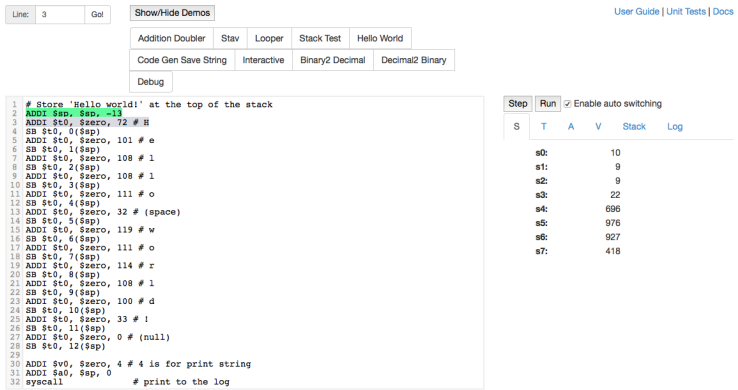
- **Registers:** locations for storing information that can be quickly accessed.

MIPS Commands



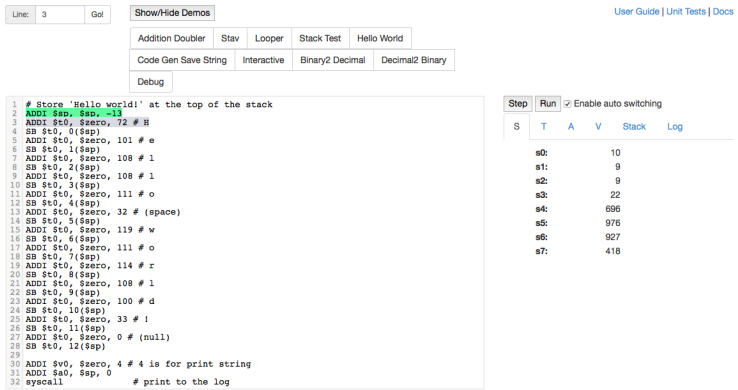
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...

MIPS Commands



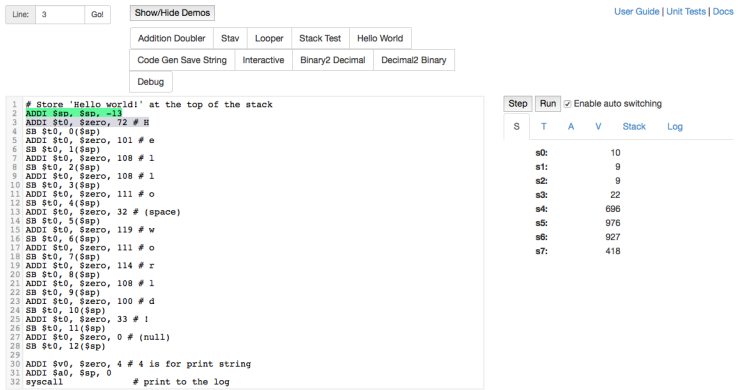
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...
- **R Instructions:** Commands that use data in the registers:

MIPS Commands



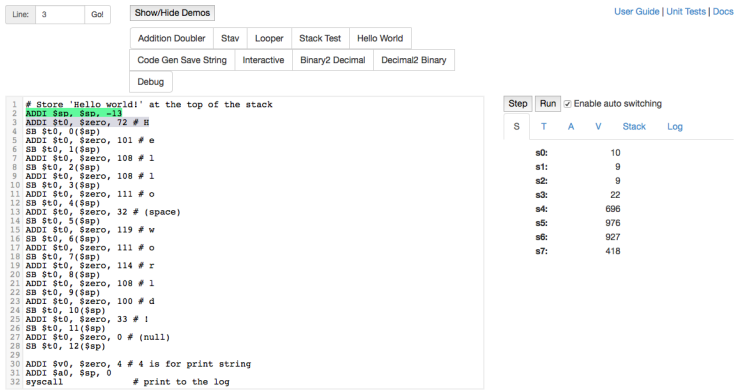
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...
- **R Instructions:** Commands that use data in the registers: `add $s1, $s2, $s3`

MIPS Commands



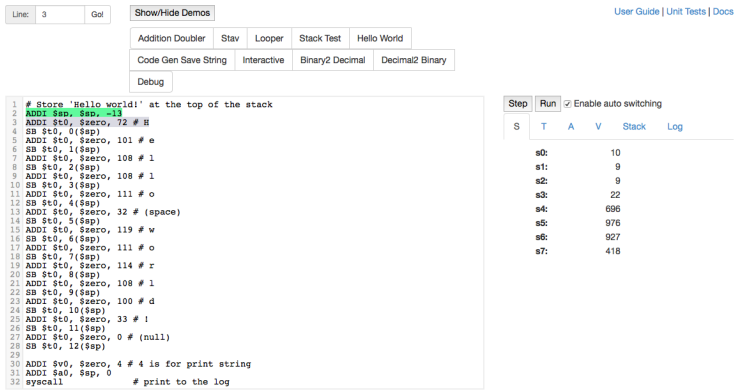
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...
- **R Instructions:** Commands that use data in the registers: `add $s1, $s2, $s3` (Basic form: `OP rd, rs, rt`)
- **I Instructions:** instructions that also use intermediate values.

MIPS Commands



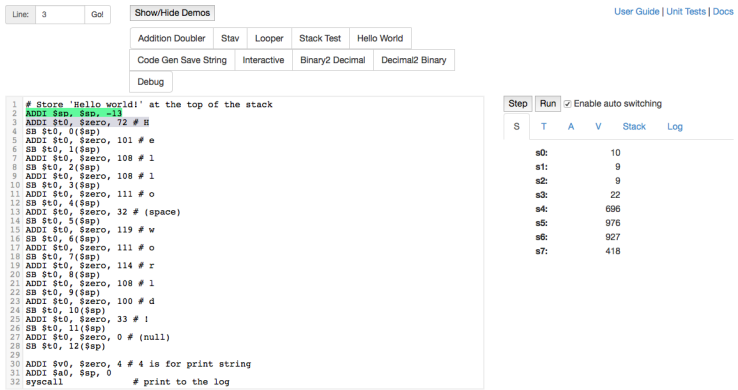
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...
- **R Instructions:** Commands that use data in the registers:
  `add $s1, $s2, $s3` (Basic form: `OP rd, rs, rt`)
- **I Instructions:** instructions that also use intermediate values.
  `addi $s1, $s2, 100`

MIPS Commands



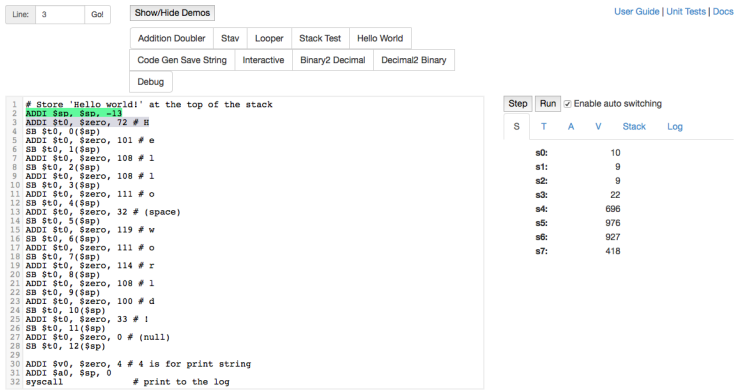
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...
- **R Instructions:** Commands that use data in the registers:
  `add $s1, $s2, $s3` (Basic form: `OP rd, rs, rt`)
- **I Instructions:** instructions that also use intermediate values.
  `addi $s1, $s2, 100` (Basic form: `OP rd, rs, imm`)
- **J Instructions:** instructions that jump to another memory location.

MIPS Commands



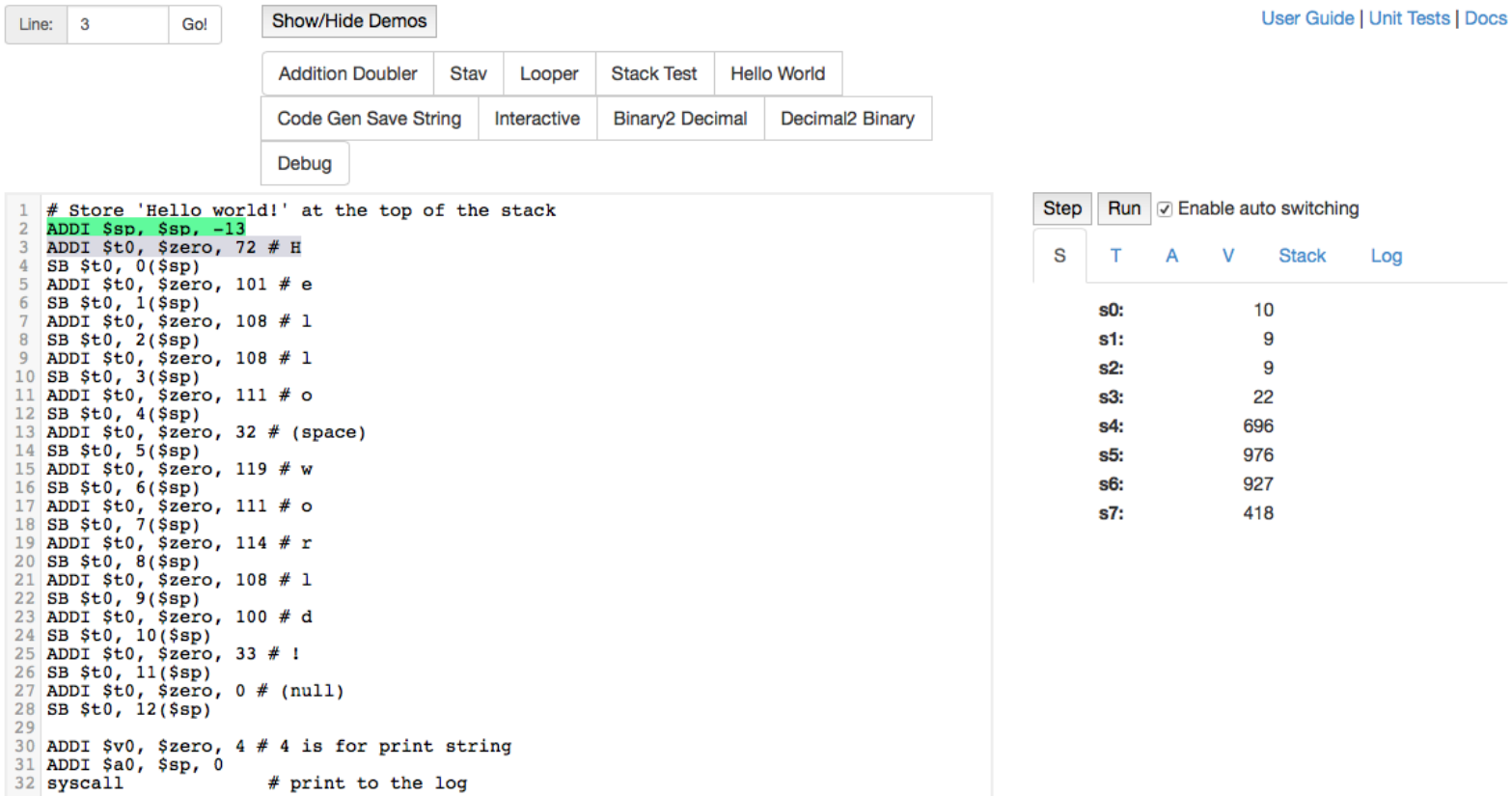
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...
- **R Instructions:** Commands that use data in the registers:
  `add $s1, $s2, $s3` (Basic form: `OP rd, rs, rt`)
- **I Instructions:** instructions that also use intermediate values.
  `addi $s1, $s2, 100` (Basic form: `OP rd, rs, imm`)
- **J Instructions:** instructions that jump to another memory location.
  `j done`

MIPS Commands



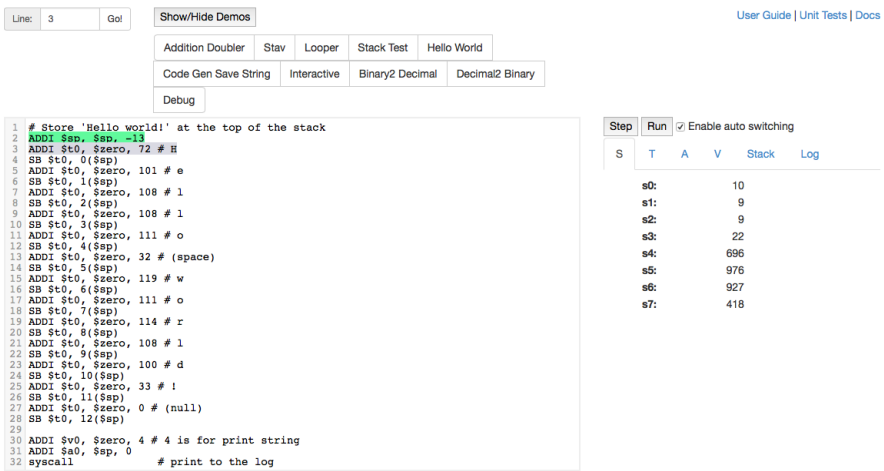
- **Registers:** locations for storing information that can be quickly accessed. Names start with '$': `$s0`, `$s1`, `$t0`, `$t1`,...
- **R Instructions:** Commands that use data in the registers:
  `add $s1, $s2, $s3` (Basic form: `OP rd, rs, rt`)
- **I Instructions:** instructions that also use intermediate values.
  `addi $s1, $s2, 100` (Basic form: `OP rd, rs, imm`)
- **J Instructions:** instructions that jump to another memory location.
  `j done` (Basic form: `OP label`)

Challenge:



Write a program that prints out the alphabet: `a b c d ... x y z`

WeMIPS



(Demo with `WeMIPS`)

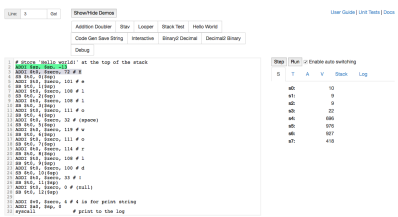
Today's Topics

- Design Patterns: Searching
- Python Recap
- Machine Language
- **Machine Language: Jumps & Loops**

- Final Exam: Format

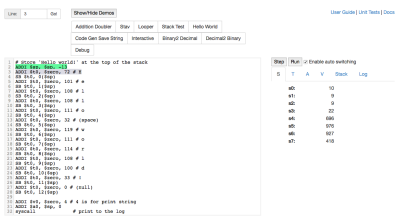
Loops & Jumps in Machine Language

- Instead of built-in looping structures like `for` and `while`, you create your own loops by "jumping" to the location in the program.

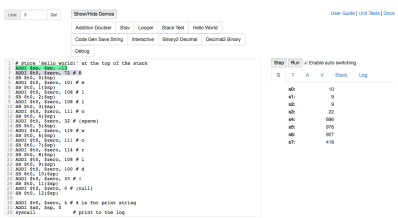


Loops & Jumps in Machine Language

- Instead of built-in looping structures like `for` and `while`, you create your own loops by "jumping" to the location in the program.
- Can indicate locations by writing **labels** at the beginning of a line.

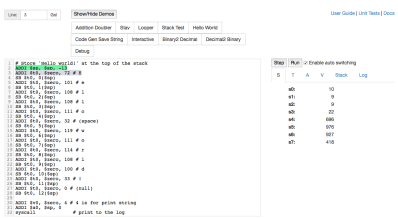


Loops & Jumps in Machine Language



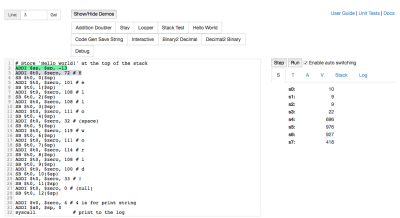
- Instead of built-in looping structures like `for` and `while`, you create your own loops by "jumping" to the location in the program.
- Can indicate locations by writing **labels** at the beginning of a line.
- Then give a command to jump to that location.

Loops & Jumps in Machine Language



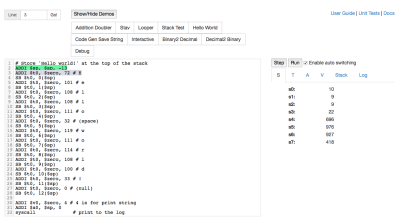
- Instead of built-in looping structures like `for` and `while`, you create your own loops by "jumping" to the location in the program.
- Can indicate locations by writing **labels** at the beginning of a line.
- Then give a command to jump to that location.
- Different kinds of jumps:

Loops & Jumps in Machine Language



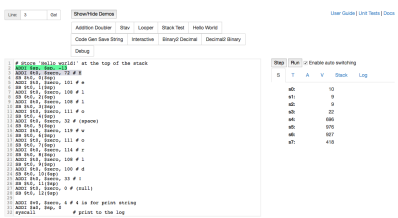
- Instead of built-in looping structures like `for` and `while`, you create your own loops by "jumping" to the location in the program.
- Can indicate locations by writing **labels** at the beginning of a line.
- Then give a command to jump to that location.
- Different kinds of jumps:
  - **Unconditional:** `j Done` will jump to the address with label `Done`.

Loops & Jumps in Machine Language



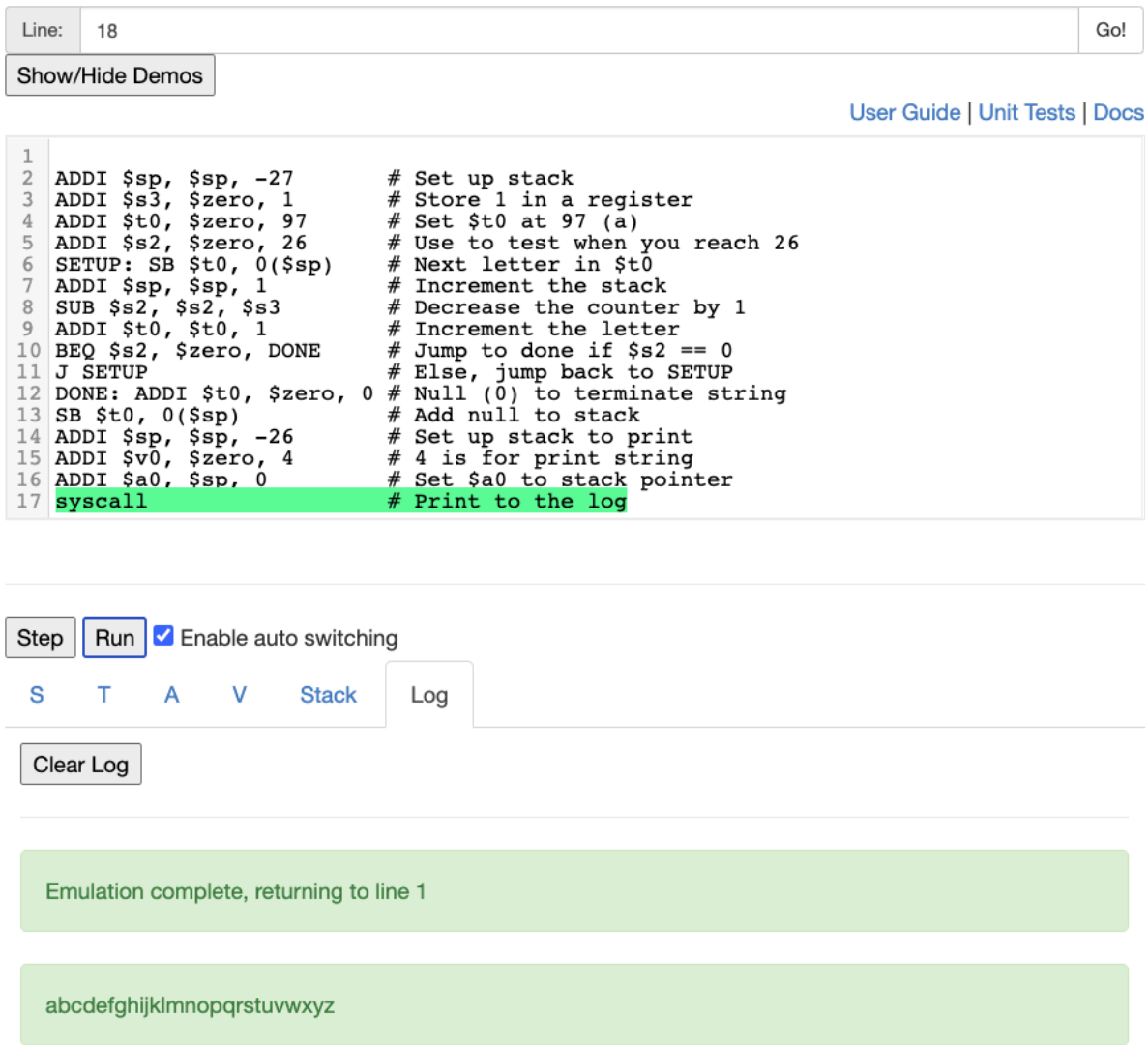
- Instead of built-in looping structures like `for` and `while`, you create your own loops by "jumping" to the location in the program.
- Can indicate locations by writing **labels** at the beginning of a line.
- Then give a command to jump to that location.
- Different kinds of jumps:
  - **Unconditional:** `j Done` will jump to the address with label `Done`.
  - **Branch if Equal:** `beq $s0 $s1 DoAgain` will jump to the address with label `DoAgain` if the registers $s0 and $s1 contain the same value.

Loops & Jumps in Machine Language



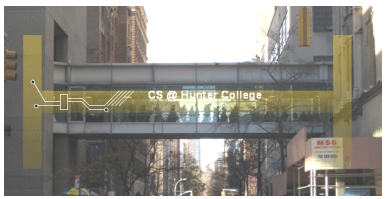
- Instead of built-in looping structures like `for` and `while`, you create your own loops by "jumping" to the location in the program.
- Can indicate locations by writing **labels** at the beginning of a line.
- Then give a command to jump to that location.
- Different kinds of jumps:
  - **Unconditional:** `j Done` will jump to the address with label `Done`.
  - **Branch if Equal:** `beq $s0 $s1 DoAgain` will jump to the address with label `DoAgain` if the registers $s0 and $s1 contain the same value.
  - See reading for more variations.

Jump Demo



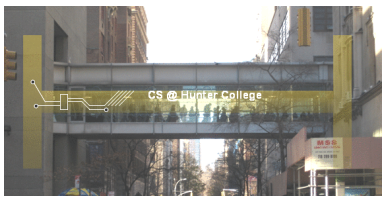
(Demo with `WeMIPS`)

Recap



- Searching through data is a common task– built-in functions and standard design patterns for this.

Recap



- Searching through data is a common task– built-in functions and standard design patterns for this.
- Programming languages can be classified by the level of abstraction and direct access to data.

Today's Topics

- Design Patterns: Searching
- Python Recap
- Machine Language
- Machine Language: Jumps & Loops
- Binary & Hex Arithmetic
- **Final Exam: Format**

Final Overview: Administration

- The exam will be administered in class, on paper.

Final Overview: Administration

- The exam will be administered in class, on paper.
- Please arrive 15 minutes earlier than the start time so there is no rush and chaos
- The exam format:

Final Overview: Administration

- The exam will be administered in class, on paper.
- Please arrive 15 minutes earlier than the start time so there is no rush and chaos
- The exam format:
  - Questions will correspond to the parts of old exams, the types of topics/concepts will be same but obviously different questions

Final Overview: Administration

- The exam will be administered in class, on paper.
- Please arrive 15 minutes earlier than the start time so there is no rush and chaos
- The exam format:
  - Questions will correspond to the parts of old exams, the types of topics/concepts will be same but obviously different questions
  - Questions are variations on the programming assignments, lab exercises, and lecture design challenges.

Final Overview: Administration

- The exam will be administered in class, on paper.
- Please arrive 15 minutes earlier than the start time so there is no rush and chaos
- The exam format:
  - Questions will correspond to the parts of old exams, the types of topics/concepts will be same but obviously different questions
  - Questions are variations on the programming assignments, lab exercises, and lecture design challenges.

Final Overview: Format

- You are allowed to prepare and bring in 1 piece of **8.5" x 11"** paper.

Final Overview: Format

- You are allowed to prepare and bring in 1 piece of **8.5" x 11"** paper.
  - With notes, examples, programs: what will help you on the exam.

Final Overview: Format

- You are allowed to prepare and bring in 1 piece of **8.5" x 11"** paper.
  - With notes, examples, programs: what will help you on the exam.
  - Best if you design/write yours since excellent way to study.

Final Overview: Format

- You are allowed to prepare and bring in 1 piece of **8.5" x 11"** paper.
  - With notes, examples, programs: what will help you on the exam.
  - Best if you design/write yours since excellent way to study.
  - Avoid scrambling through web searches and waste time during the exam.

Final Overview: Format

- You are allowed to prepare and bring in 1 piece of **8.5" x 11"** paper.
  - With notes, examples, programs: what will help you on the exam.
  - Best if you design/write yours since excellent way to study.
  - Avoid scrambling through web searches and waste time during the exam.
- Past exams available on webpage (includes answer keys).

Class Reminders!

Before next lecture, don't forget to:

- Review this class's Lecture and Lab 11!

Class Reminders!

Before next lecture, don't forget to:

- Review this class's Lecture and Lab 11!
- Batch 9 is due TOMORROW 6pm!!!