

# CSci 127: Introduction to Computer Science



[hunter.cuny.edu/csci](http://hunter.cuny.edu/csci)

# Announcements



- Spring Break starts Friday!

# Announcements



- Spring Break starts Friday!
- No CUNY classes:  
*Friday, 19 April through Sunday, 28 April.*

# Announcements



- Spring Break starts Friday!
- No CUNY classes:  
*Friday, 19 April through Sunday, 28 April.*
- CS Survey: Anna Whitney  
Google Storage Infrastructure Team

# Today's Topics



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

# Today's Topics



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

# Python & Circuits Review: 10 Weeks in 10 Minutes



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

# Week 1: print(), loops, comments, & turtles

# Week 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

# Week 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:

The screenshot shows a Python code editor interface. The left pane displays the code file `main.py` with the following content:

```
1 #A program that demonstrates turtles stamping
2
3 import turtle
4
5 taylor = turtle.Turtle()
6 taylor.color("purple")
7 taylor.shape("turtle")
8
9 for i in range(6):
10     taylor.forward(100)
11     taylor.stamp()
12     taylor.left(60)
```

The right pane has two tabs: `Result` and `Instructions`. The `Result` tab shows the output of the program, which is a regular hexagon drawn in purple. Each vertex of the hexagon has a small purple star-like stamp.

# Week 2: variables, data types, more on loops & range()

## Week 2: variables, data types, more on loops & range()

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

## Week 2: 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

## Week 2: 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

## Week 2: 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

## Week 2: 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

## Week 2: 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']

## Week 2: 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.

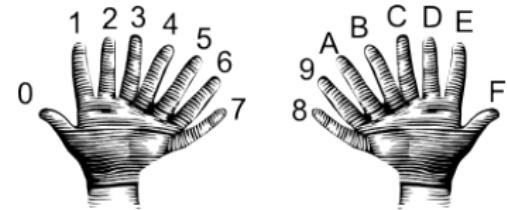
## Week 2: variables, data types, more on loops & range()

- 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(x)  
12  
13 for c in "ABCD":  
14     print(c)
```

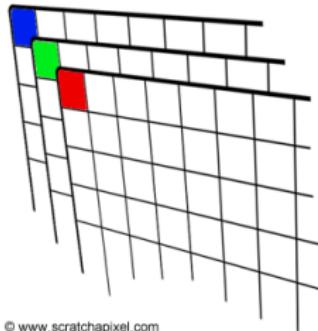
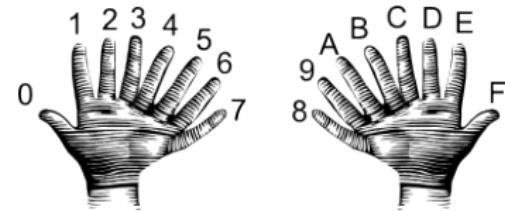
# Week 3: colors, hex, slices, numpy & images

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



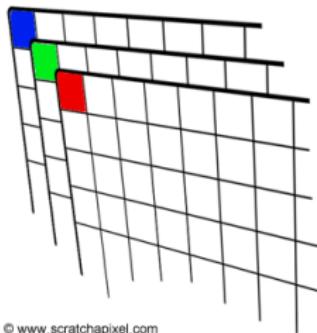
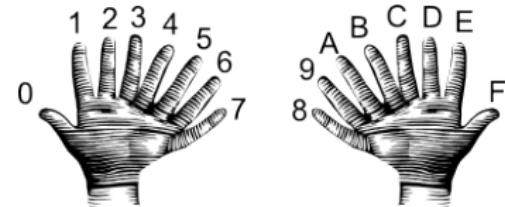
# Week 3: colors, hex, slices, numpy & images

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



# Week 3: 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]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Week 4: design problem (cropping images) & decisions



# Week 4: design problem (cropping images) & decisions



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

# Week 4: 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.

# Week 4: 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.

## Week 4: 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')
```

# Week 5: 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!")
```

# Week 5: 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	and	in2	returns:
False	and	False	False
False	and	True	False
True	and	False	False
True	and	True	True



# Week 6: structured data, pandas, & more design

Source: [https://en.wikipedia.org/wiki/Demographics\\_of\\_New\\_York\\_City](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  
1690,4937,1037,727,788,1100  
1771,21843,3623,1,2847,28423  
1790,33131,4549,6159,1781,3827,49447  
1800,60515,5740,6442,1755,4543,75955  
1810,65541,6250,6840,2035,49734  
1820,123704,11187,8246,2792,6135,152056  
1830,202589,20535,9049,3023,7082,242278  
1840,312110,11013,14045,5346,10965,391114  
1850,35544,12850,18850,5346,10965,50115  
1860,813469,279122,32903,23593,25492,174777  
1870,942292,419921,45468,37393,33029,1479103  
1880,1164473,59945,5653,51980,33029,1911801  
1890,1367711,70000,65000,5653,51980,2411314  
1900,185093,116582,152999,200567,67921,2437202  
1910,223342,1634351,284641,430980,8569,4766803  
1920,2211103,2018354,446070,720201,11651,501488  
1930,1867128,1796128,2230936,1891325,158245,4930446  
1940,1889924,2498285,1297634,1394711,174441,7454995  
1950,1960101,2738175,1550849,1451277,191555,7991957  
1960,1690331,1809239,1809239,1451277,191555,7981984  
1970,1539231,1865705,1865705,1471701,195443,7981984  
1980,1426285,2230936,1891325,1168972,352121,7071639  
1990,1487536,2300664,1951598,1203789,378977,7322564  
2000,1537195,2485326,2229379,1332650,419782,8080879  
2010,1583873,2504705,2216722,1385108,447515,8175133  
2015,1444518,2436733,2339150,1454446,474558,8059405

nycHistPop.csv

In Lab 6

# Week 6: 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](https://en.wikipedia.org/wiki/Demographics_of_New_York_City),  
All population figures are consistent with present-day boundaries.....  
Five census after the consolidation of the five boroughs.....  
.....  
Year,Manhattan,Brooklyn,Queens,Bronx,Staten Island,Total  
1890,4937,2037,727,788,1000  
1891,21843,3623,2847,28423  
1790,33131,4549,6159,1781,3827,49447  
1800,60515,5740,6442,1755,4543,7595  
1810,63545,5540,6242,1755,4543,7595  
1820,123704,11487,8246,2792,6135,152056  
1830,202589,20535,9049,3023,7082,242278  
1840,312110,18013,14049,5346,10965,391114  
1850,35549,12891,12891,12891,12891,12891  
1860,813469,279122,32903,23593,25492,174777  
1870,942292,419921,45468,37393,33029,1479103  
1880,1164473,59945,5653,51980,33029,1911803  
1890,1367711,66582,61582,5653,51980,33029,1911803  
1900,185093,116582,152999,200567,67621,2437202  
1910,223342,1634351,2841,430980,8569,4766803  
1920,22103,2018354,44607,44607,73201,116582,51980  
1930,66713,116582,152999,200567,67621,2437202  
1940,1889924,2498285,1297634,1394711,174441,7454995  
1950,1960101,2738175,1550949,1451277,191555,7991957  
1960,1660101,2738175,1550949,1451277,191555,7991957  
1970,1659331,2460705,1472705,1472705,135443,7981984  
1980,1426285,2230936,1891325,1168972,352121,7071639  
1990,1487536,2300664,1951598,1203789,378977,7322564  
2000,1537195,2485326,2229379,1332450,419782,8080879  
2010,1583873,2504705,2277722,1385108,419782,8175133  
2015,1444918,2436733,2339150,1459446,474558,8059405

nycHistPop.csv

In Lab 6

# Week 6: 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,Bronx,Brooklyn,Queens,Bronx,Staten Island,Totals  
1690,4937,2037,727,788,100  
1771,21843,3623,2847,28423  
1790,33131,4549,6159,1781,3827,49447  
1800,60515,5740,6442,1755,4543,75955  
1810,65541,6240,6842,1755,4543,79334  
1820,123704,11187,8246,2792,6135,152056  
1830,202589,20535,9049,3032,7082,242278  
1840,311510,19013,14081,5346,10965,391114  
1850,35544,21891,18891,5346,10965,391115  
1860,813469,279122,32903,23593,25492,174777  
1870,942292,419921,45468,37393,33029,1479103  
1880,1164473,59943,5653,51980,33029,1911801  
1890,1367000,70000,65000,51980,33029,2140134  
1900,1850593,116582,152999,200567,67621,2437202  
1910,2233142,1634351,2841,430980,8569,476683  
1920,2210110,2018354,44601,44601,73201,11651,50048  
1930,1867112,2000000,579128,579128,55821,4930446  
1940,1889924,2498285,1297634,1394711,174441,7454995  
1950,1960101,2738175,1550949,1451277,191555,7991957  
1960,1690101,2738175,1550949,1451277,191555,7981984  
1970,1539231,2467071,1472701,1235443,7071646  
1980,1426285,2230936,1891325,1168972,352121,7071639  
1990,1487536,2300664,1951598,1203789,7322564  
2000,1537195,2485326,2223379,1332450,419728,8080879  
2010,1583873,2504705,2277722,1385108,8175133  
2015,1444018,2646733,2339150,1459446,474558,8059405
```

nycHistPop.csv

In Lab 6

# Week 6: 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,Population  
1690,4937,2017,...,727,7181  
1771,21843,36232,...,2847,28423  
1790,33131,4549,6159,1781,3827,49447  
1800,60515,5740,6442,1755,4543,75955  
1810,70000,62000,68000,1800,49734  
1820,123704,11187,8246,2792,6135,152056  
1830,20589,20535,9049,3023,7082,242278  
1840,311510,11013,14000,5348,10965,391114  
1850,35549,128000,128000,128000,128000,15  
1860,813469,279122,32903,23593,25492,174777  
1870,942292,419921,45468,37393,33029,1479103  
1880,1164473,59940,5653,51980,33091,1911801  
1890,1370000,650000,650000,650000,650000,134  
1900,1850093,116582,152999,200567,67621,2437202  
1910,2331842,1634351,2841,430980,8569,476683  
1920,2216103,2018354,44601,72021,11671,50048  
1930,1967128,1796128,1796128,1796128,1796128,4930446  
1940,1889924,2698285,1297634,1394711,174441,7454995  
1950,1960101,2738175,1550949,1451277,191555,7991957  
1960,1690000,2319319,1809000,1690000,1690000,781984  
1970,1539231,2467011,1871473,1871701,185443,798462  
1980,1426285,2230936,1891325,1168972,352121,7071639  
1990,1487536,2300664,1951598,1203789,739977,7322564  
2000,1537195,2485326,2223379,1332450,419728,8080879  
2010,1583873,2504705,2271722,1385108,474558,8175133  
2015,1444018,2636733,2339150,1459446,474558,8056405
```

nycHistPop.csv

In Lab 6

# Week 6: structured data, pandas, & more design

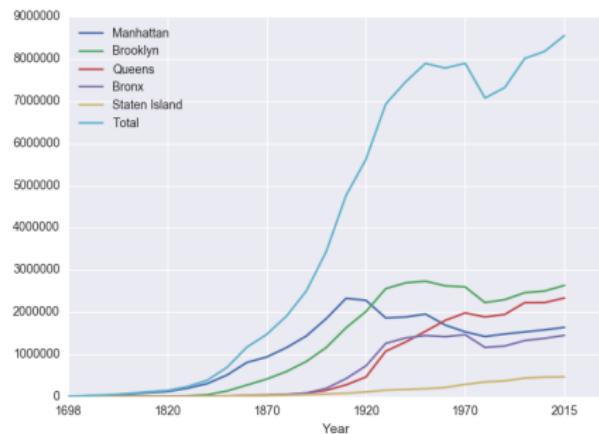
```
import matplotlib.pyplot as plt  
import pandas as pd
```

```
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,Population  
1699,4937,2017,...,727,7181  
1771,21843,36231,...,2847,28423  
1790,33131,4549,6159,1781,3827,49447  
1800,60515,5740,6442,1755,4543,75955  
1810,70000,6350,7000,1800,5000,89734  
1820,123704,11187,8246,2792,6135,152056  
1830,202589,20535,9049,3023,7082,242278  
1840,312110,20113,14000,5348,10965,391114  
1850,455441,21800,18500,5800,12000,51014  
1860,813469,279122,32903,23593,25492,174777  
1870,942292,419921,45468,37393,33029,1479103  
1880,1164473,59943,5653,51980,33091,1911801  
1890,1400000,720000,68000,63000,58000,210000  
1900,1850093,116582,152999,200507,67921,2437202  
1910,233142,1634351,2841,430980,8569,476683  
1920,2210103,2018354,44601,732013,11651,50048  
1930,2667103,2467128,35254,35254,5820,4930446  
1940,1889924,2690285,1297634,1394711,174441,7454995  
1950,1960101,2738175,1550849,1451277,191555,7091957  
1960,1696000,2731935,1489000,1400000,1300000,7081984  
1970,1539231,2467011,1472701,135443,7084640  
1980,1426285,2230936,11891325,1168972,352121,7071639  
1990,1487536,2300664,1951598,1203789,378977,7322564  
2000,1537195,2485326,2229379,1332450,413728,8080879  
2010,1583873,2504705,2216722,1385108,413728,8175133  
2015,1444518,2636733,2339150,1459446,474558,8059405
```

nycHistPop.csv

In Lab 6



# Week 7: functions

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

```
#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()
```

# Week 7: 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.

# Week 7: 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()`

# Week 7: 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:

# Week 7: 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")`

# Week 7: 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,

# Week 7: 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.

# Week 8: 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)
```

# Week 8: 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).

# Week 8: 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**.

# Week 8: 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**

# Week 8: 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.

# Week 8: 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)
print('Actual Parameters')

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.

# Week 9: 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)
```

# Week 10: 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)
```

# Week 10: 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)
```

# Week 10: 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.

# Week 10: 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.`

# Week 10: 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: 10 Weeks in 10 Minutes



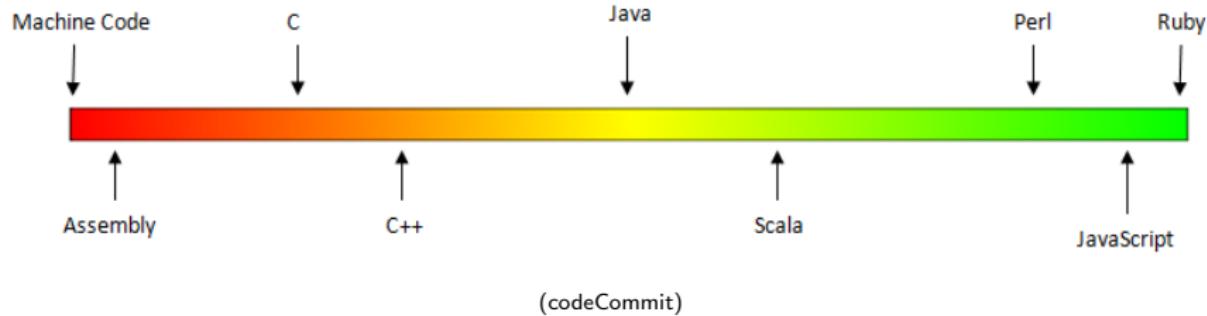
- Input/Output (I/O): `input()` and `print()`; pandas for CSV files
- Types:
  - ▶ Primitive: `int`, `float`, `bool`, `string`;
  - ▶ Container: lists (but not dictionaries/hashes or tuples)
- Objects: turtles (used but did not design our own)
- Loops: definite & indefinite
- Conditionals: `if-elif-else`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

# Today's Topics



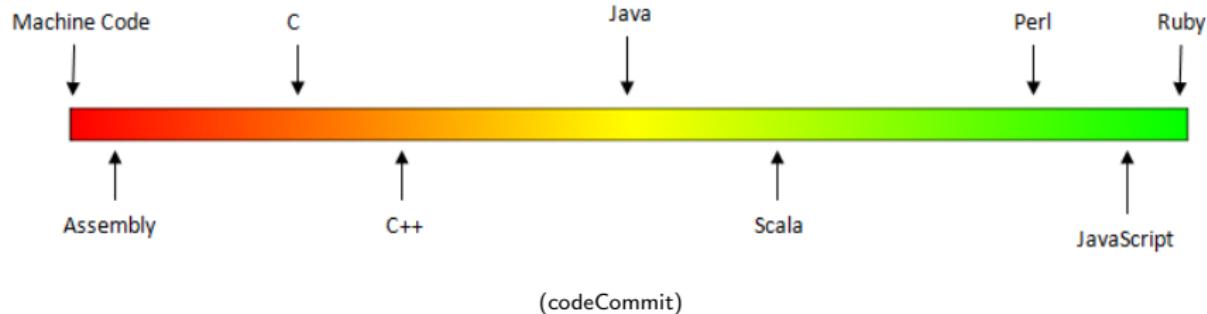
- Python Recap
- **Machine Language**
- Machine Language: Jumps & Loops
- Design Patterns: Searching
- CS Survey

# Low-Level vs. High-Level Languages



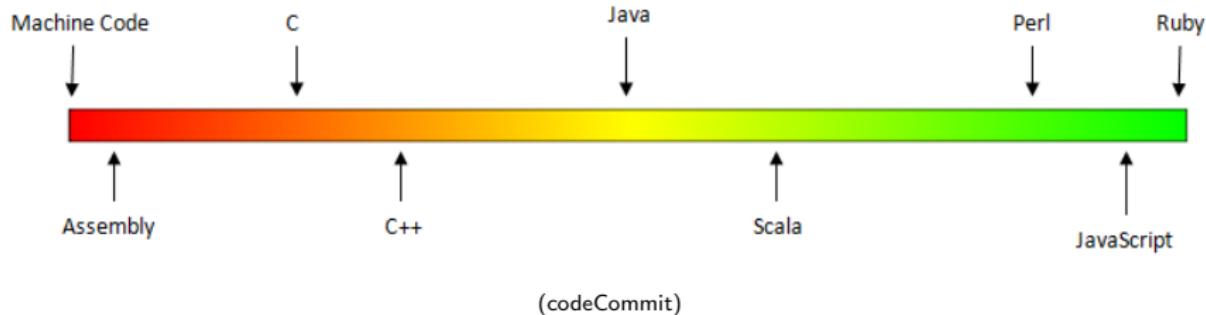
- Can view programming languages on a continuum.

# Low-Level vs. High-Level Languages



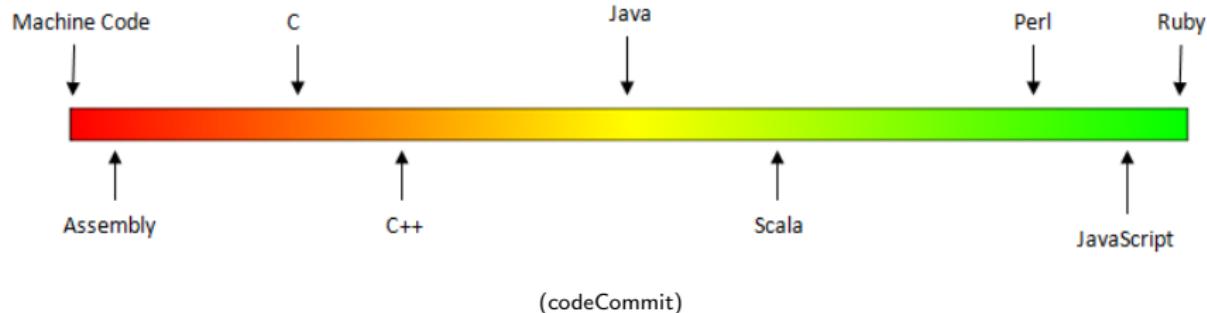
- Can view programming languages on a continuum.
- Those that directly access machine instructions & memory and have little abstraction are **low-level languages**

# Low-Level vs. High-Level Languages



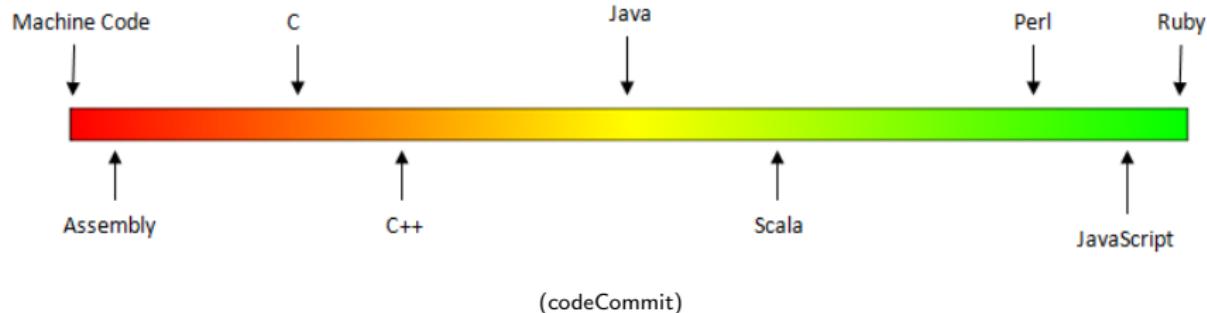
- Can view programming languages on a continuum.
- Those that directly access machine instructions & memory and have little abstraction are **low-level languages** (e.g. machine language, assembly language).

# Low-Level vs. High-Level Languages



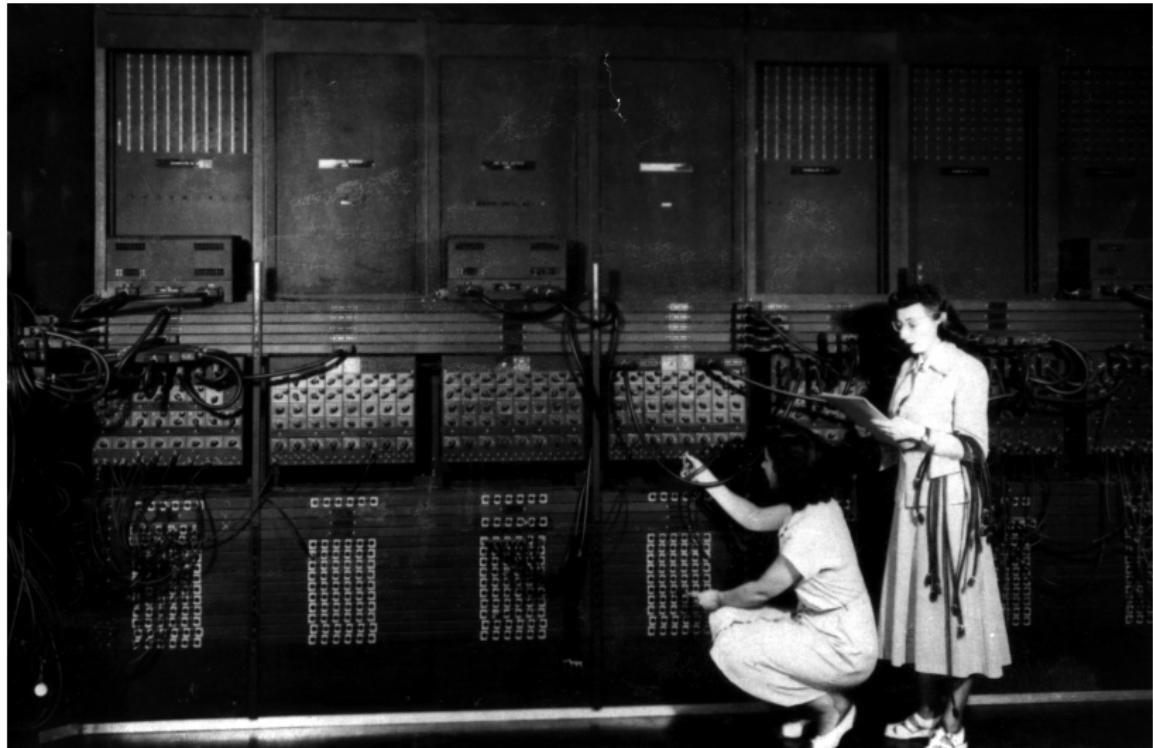
- Can view programming languages on a continuum.
- Those that directly access machine instructions & memory and have little abstraction are **low-level languages** (e.g. machine language, assembly language).
- Those that have strong abstraction (allow programming paradigms independent of the machine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



- 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.

# Machine Language



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

# Machine Language

```
I FOX 12:01a 23- 1
A 002000 C2 30      REP #$30
A 002002 18          CLC
A 002003 F8          SED
A 002004 A9 34 12    LDA #$1234
A 002007 69 21 43    ADC #$4321
A 00200A 8F 03 7F 01 STA $017F03
A 00200E D8          CLD
A 00200F E2 30      SEP #$30
A 002011 00          BRK
A 2012

r
PB PC  NUMxDIZC .A .X .Y SP DP DB
; 00 E012 00110000 0000 0000 0002 CFFF 0000 00
g 2000

BREAK

PB PC  NUMxDIZC .A .X .Y SP DP DB
; 00 2013 00110000 5555 0000 0002 CFFF 0000 00
m 7f03 7f03
>007F03 55 55 00 00 00 00 00 00 00 00 00 00 00 00 00:UU .....
```

(wiki)

# Machine Language

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

The screenshot shows a terminal window with two panes. The left pane displays assembly code:

```
R 00200000 C2 30 [4] 0xa 2f-]
R 00200002 1B CB
R 00200003 FB CD
R 00200004 69 34 12 LDa #1234
R 00200007 69 21 43 LDc #4321
R 00200008 8F 03 7F 01 STA #17F03
R 0020000E 8F 03 CLD
R 00200010 E2 30 SER #30
R 00200011 80 SWR
R 20012

P# PC M[0x00200000..0x00200011] A .X .Y SP DP R#
: 00 E012 00100000 0000 0000 0002 CFFF 0000 00
& 20000

BREAK

P# PC M[0x00200000..0x00200011] A .X .Y SP DP R#
: 00 E012 00100000 5555 0000 0002 CFFF 0000 00
& 20000

>00200011: 55 55 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The right pane shows the memory dump:

```
M[0x00200000..0x00200011] A .X .Y SP DP R#
: 00 E012 00100000 5555 0000 0002 CFFF 0000 00
& 20000

>00200011: 55 55 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

(wiki)

# Machine Language

- 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.



The screenshot shows a window titled 'File' with tabs for 'Assembly' and 'Binary'. The assembly code area contains:

```
    C2 38    REP #38
    0020021B    CLD
    002003FB    SEI
    002004E9 34 12    LDA #1234
    00200769 21 43    ADC #4324
    002008BF 03 7F 01    STA #17F03
    002009E8 21 30    CLD
    00200AE9 34 38    SEI #38
    00201180    BRA
    002012    
```

The binary code area below shows the assembly code mapped to memory addresses:

PC	Memory Address	Value
0020021B	00110000 0000 0000 0002 CFFF 0000 00	
002003FB	00110000 5555 0000 0002 CFFF 0000 00	
002004E9	00110000 7FFF 0000 0002 CFFF 0000 00	
00200769	00110000 55 55 00 00 00 00 00 00 00 00 00 00 00 00 00	
002008BF	00110000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
002009E8	00110000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00200AE9	00110000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00201180	00110000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
002012	00110000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

(wiki)

# Machine Language

The screenshot shows a computer screen with a window titled "WeMIPS" containing assembly code and its corresponding binary output. The assembly code includes instructions like REP #838, LD \$T0, ST \$T0, and various arithmetic and memory access operations. Below the assembly code, the binary representation is shown in columns for PC, R/M, A, X, Y, SP, DP, and R/W. At the bottom, there is a "BREAK" button.

```
    C2 3B    REP #838
A 002002 1B    CLD
A 002003 FB    SEI
A 002004 69 34 12    LDA #1234
A 002007 69 21 43    ADC #4321
A 002008 BF 03 7F 01    STA #017F03
A 00200E E9 30    CLD
A 002010 69 38    SEI #838
A 002011 BB    BRK
A 2012

PB PC    M/Hm012C A   X   Y   SP   DP   R/W
: 00 E912 00110000 0000 0000 0002 CFFF 0000 00
& 20000

BREAK
```

(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.



# “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 # i
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	<input checked="" type="checkbox"/> 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

The screenshot shows the WeMIPS debugger interface. At the top, there are tabs for "Live", "3", "Data", and "Show While Device". Below these are navigation buttons for "Addition Doubler", "Stax", "Looper", "Stack Test", and "Hello World". There are also links for "Code Gen Save String", "Interactive", "Binary2 Decimal", and "Decimal2 Binary". A "Debug" button is also present.

The main area contains assembly code for printing "Hello world!" to the screen. The code includes instructions like `ADDI` for loading immediate values (72, 101, etc.) into registers, `ADDI` for setting up memory pointers (72, 101), and `MOVI` for loading the string "Hello world!". The assembly code is as follows:

```
# Store 'Hello world!' at the top of the stack
1 MOVI $t0, $zero, 72 # N
2 ADDI $t1, $zero, 101 # e
3 MOVI $t2, $zero, 101 # m
4 ADDI $t3, $zero, 101 # l
5 MOVI $t4, $zero, 108 # o
6 MOVI $t5, $zero, 108 # l
7 ADDI $t6, $zero, 108 # i
8 ADDI $t7, $zero, 108 # n
9 ADDI $t8, $zero, 108 # d
10 ADDI $t9, $zero, 33 # !
11 ADDI $t10, $zero, 0 # null
12 ADDI $t11, $zero, 0 # null
13 ADDI $t12, $zero, 0 # null
14 ADDI $t13, $zero, 0 # null
15 ADDI $t14, $zero, 0 # null
16 ADDI $t15, $zero, 0 # null
17 ADDI $t16, $zero, 0 # null
18 ADDI $t17, $zero, 0 # null
19 ADDI $t18, $zero, 0 # null
20 ADDI $t19, $zero, 0 # null
21 ADDI $t20, $zero, 0 # null
22 ADDI $t21, $zero, 0 # null
23 ADDI $t22, $zero, 0 # null
24 ADDI $t23, $zero, 0 # null
25 ADDI $t24, $zero, 0 # null
26 ADDI $t25, $zero, 0 # null
27 ADDI $t26, $zero, 0 # null
28 ADDI $t27, $zero, 0 # null
29 ADDI $t28, $zero, 0 # null
30 ADDI $t29, $zero, 0 # null
31 ADDI $t30, $zero, 0 # null
32 ADDI $t31, $zero, 0 # null
33 MOVI $s0, $zero, 4 # 4 is for print string
34 ADDI $s0, $sp, 0 # point to the log
35 syscall
```

To the right of the assembly code is a table showing register values:

Step	T	A	V	Stack	Log
S					
s0	10				
s1	9				
s2	9				
s3	22				
s4	995				
s5	976				
s6	977				
s7	419				

(Demo with WeMIPS)

# MIPS Commands

The screenshot shows a MIPS assembly debugger interface. On the left, there is a text area containing assembly code. On the right, there is a register dump table.

**Assembly Code:**

```
1 # Show "Hello world" at the top of the stack
2 .data
3 .text
4 .globl _start
5 .text
6 addi $t0, $zero, 101 # a
7 addi $t0, $zero, 101 # a
8 addi $t0, $zero, 100 # 1
9 addi $t0, $zero, 100 # 1
10 addi $t0, $zero, 100 # 1
11 addi $t0, $zero, 100 # 1
12 addi $t0, $zero, 100 # 1
13 addi $t0, $zero, 100 # 1
14 addi $t0, $zero, 100 # 1
15 addi $t0, $zero, 100 # 1
16 addi $t0, $zero, 100 # 1
17 addi $t0, $zero, 100 # 1
18 addi $t0, $zero, 100 # 1
19 addi $t0, $zero, 100 # 1
20 addi $t0, $zero, 100 # 1
21 addi $t0, $zero, 100 # 1
22 addi $t0, $zero, 100 # d
23 addi $t0, $zero, 100 # d
24 addi $t0, $zero, 100 # d
25 addi $t0, $zero, 100 # d
26 addi $t0, $zero, 100 # d
27 addi $t0, $zero, 100 # d
28 addi $t0, $zero, 100 # d
29 addi $t0, $zero, 100 # d
30 addi $t0, $zero, 100 # d
31 addi $t0, $zero, 100 # d
32 syscall
```

**Registers:**

S	T	A	V	Stack	Log
\$t0	10				
\$t1	9				
\$t2	8				
\$t3	22				
\$t4	60				
\$t5	61				
\$t6	807				
\$t7	418				

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

# MIPS Commands

The screenshot shows a MIPS assembly debugger interface. At the top, there are tabs for "Show/Hide Demo", "User Guide", "Unit Tests", and "Docs". Below the tabs are several menu options: "Addition Counter", "Btav", "Looper", "Stack Test", "Hello World", "Code Gen Save String", "Interactive", "Binary Decimal", "Decimal Binary", and "Debug". The "Debug" tab is currently selected.

The main area displays assembly code:

```
1 # Shows "Hello world" at the top of the stack
2 .text
3 .globl _start
4 _start:
5 addi $t0,$zero,$111 # a
6 addi $t1,$zero,$111 # b
7 addi $t2,$zero,$111 # c
8 addi $t3,$zero,$111 # d
9 addi $t4,$zero,$111 # e
10 addi $t5,$zero,$111 # f
11 addi $t6,$zero,$111 # g
12 addi $t7,$zero,$111 # h
13 addi $t8,$zero,$111 # i
14 addi $t9,$zero,$111 # j
15 addi $t10,$zero,$111 # k
16 addi $t11,$zero,$111 # l
17 addi $t12,$zero,$111 # m
18 addi $t13,$zero,$111 # n
19 addi $t14,$zero,$111 # o
20 addi $t15,$zero,$111 # p
21 addi $t16,$zero,$111 # q
22 addi $t17,$zero,$111 # r
23 addi $t18,$zero,$111 # s
24 addi $t19,$zero,$111 # t
25 addi $t20,$zero,$111 # u
26 addi $t21,$zero,$111 # v
27 addi $t22,$zero,$111 # w
28 addi $t23,$zero,$111 # x
29 addi $t24,$zero,$111 # y
30 addi $t25,$zero,$111 # z
31 addi $t26,$zero,$111 # {rval}
32 addi $t27,$zero,$111 # }rval
33 addi $t28,$zero,$111 # print to the log
34 addi $t29,$zero,$111 # syscall
```

To the right of the code, there is a register table titled "Registers" with columns for S, T, A, V, Stack, and Log. The register values are as follows:

S	T	A	V	Stack	Log
\$0	111				
\$1	9				
\$2	22				
\$3	66				
\$4	67				
\$5	807				
\$6	418				

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

# MIPS Commands

The screenshot shows the ShowFide Demo interface. At the top, there are tabs for User, ShowFide Demo, and other options like Addition, Counter, IfElse, Looper, StackTest, Hello World, CodeGen, SaveString, Interactive, Binary, Decimal, and Debug. Below the tabs is a text area containing assembly code:

```
1 # Shows "Hello world" at the top of the stack
2 .data
3 msg: .asciiz "Hello world\n"
4 .text
5 addi $t0, $zero, $111 # a
6 addi $t1, $zero, $111 # b
7 addi $t2, $zero, $111 # c
8 addi $t3, $zero, $111 # d
9 addi $t4, $zero, $111 # e
10 addi $t5, $zero, $111 # f
11 addi $t6, $zero, $111 # g
12 addi $t7, $zero, $111 # h
13 addi $t8, $zero, $111 # i
14 addi $t9, $zero, $111 # j
15 addi $t10, $zero, $111 # k
16 addi $t11, $zero, $111 # l
17 addi $t12, $zero, $111 # m
18 addi $t13, $zero, $111 # n
19 addi $t14, $zero, $111 # o
20 addi $t15, $zero, $111 # p
21 addi $t16, $zero, $111 # q
22 addi $t17, $zero, $111 # r
23 addi $t18, $zero, $111 # s
24 addi $t19, $zero, $111 # t
25 addi $t20, $zero, $111 # u
26 addi $t21, $zero, $111 # v
27 addi $t22, $zero, $111 # w
28 addi $t23, $zero, $111 # x
29 addi $t24, $zero, $111 # y
30 addi $t25, $zero, $111 # z
31 addi $t26, $zero, 4 # a is for print string
32 addi $t27, $zero, 5 # print to the log
33 syscall
```

Below the code is a register dump table:

S	T	A	V	Stack	Log
\$0				11	
\$1				9	
\$2				8	
\$3				22	
\$4				66	
\$5				67	
\$6				807	
\$7				418	

- **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

The screenshot shows a MIPS assembly debugger interface. The top navigation bar includes 'Show/Hide Demo' and links to 'User Guide', 'Unit Tests', and 'Docs'. Below the bar are tabs for 'Addition', 'Decimal', 'Hex', 'Looper', 'Stack Test', and 'Hello World'. The 'Hello World' tab is selected, showing the assembly code:

```
# Show "Hello world" at the top of the stack
1 L:      .text
2 .globl _start
3 .type _start, @function
4 _start: .text
5 addi   $t0, $zero, 101 # $t0 = 101
6 addi   $t1, $zero, 100 # $t1 = 100
7 addi   $t2, $zero, 100 # $t2 = 100
8 addi   $t3, $zero, 100 # $t3 = 100
9 addi   $t4, $zero, 100 # $t4 = 100
10 addi   $t5, $zero, 100 # $t5 = 100
11 addi   $t6, $zero, 100 # $t6 = 100
12 addi   $t7, $zero, 100 # $t7 = 100
13 addi   $t8, $zero, 100 # $t8 = 100
14 addi   $t9, $zero, 100 # $t9 = 100
15 addi   $t10, $zero, 100 # $t10 = 100
16 addi   $t11, $zero, 100 # $t11 = 100
17 addi   $t12, $zero, 100 # $t12 = 100
18 addi   $t13, $zero, 100 # $t13 = 100
19 addi   $t14, $zero, 100 # $t14 = 100
20 addi   $t15, $zero, 100 # $t15 = 100
21 addi   $t16, $zero, 100 # $t16 = 100
22 addi   $t17, $zero, 100 # $t17 = 100
23 addi   $t18, $zero, 100 # $t18 = 100
24 addi   $t19, $zero, 100 # $t19 = 100
25 addi   $t20, $zero, 100 # $t20 = 100
26 addi   $t21, $zero, 100 # $t21 = 100
27 addi   $t22, $zero, 0 # ($t22 = null)
28 addi   $t23, $zero, 100 # $t23 = 100
29 addi   $t24, $zero, 100 # $t24 = 100
30 addi   $t25, $zero, 100 # $t25 = 100
31 addi   $t26, $zero, 100 # $t26 = 100
32 addi   $t27, $zero, 4 # $t27 = 4 for print string
33 addi   $t28, $zero, 5 # $t28 = 5 for log
34 syscall
```

The bottom status bar shows assembly ('S'), assembly ('T'), assembly ('V'), Stack, and Log buttons.

- **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

The screenshot shows the ShowMe MIPS simulator interface. At the top, there are tabs for "ShowMe Demo", "User Guide", "Unit Tests", and "Docs". Below the tabs are buttons for "Addition Counter", "Btav", "Looper", "Stack Test", and "Hello World". Further down are buttons for "Code Gen Save String", "Interactive", "Binary Decimal", "Decimal Binary", and "Debug". A "Run" button with a play icon is also present. The main area contains assembly code and a register dump table.

Assembly code:

```
# Shows "Hello world" at the top of the stack
1    .data
2    msg: .asciiz "Hello world\n"
3
4    .text
5    .globl _start
6
7    _start:
8        addi $t0, $zero, 100 # $t0 = 100
9        addi $t1, $zero, 100 # $t1 = 100
10       addi $t2, $zero, 100 # $t2 = 100
11       addi $t3, $zero, 100 # $t3 = 100
12       addi $t4, $zero, 100 # $t4 = 100
13       addi $t5, $zero, 100 # $t5 = 100
14       addi $t6, $zero, 100 # $t6 = 100
15       addi $t7, $zero, 100 # $t7 = 100
16       addi $t8, $zero, 100 # $t8 = 100
17       addi $t9, $zero, 100 # $t9 = 100
18       addi $t10, $zero, 100 # $t10 = 100
19       addi $t11, $zero, 100 # $t11 = 100
20       addi $t12, $zero, 100 # $t12 = 100
21       addi $t13, $zero, 100 # $t13 = 100
22       addi $t14, $zero, 100 # $t14 = 100
23       addi $t15, $zero, 100 # $t15 = 100
24       addi $t16, $zero, 100 # $t16 = 100
25       addi $t17, $zero, 100 # $t17 = 100
26       addi $t18, $zero, 100 # $t18 = 100
27       addi $t19, $zero, 100 # $t19 = 100
28       addi $t20, $zero, 100 # $t20 = 100
29       addi $t21, $zero, 100 # $t21 = 100
30       addi $t22, $zero, 100 # $t22 = 100
31       addi $t23, $zero, 100 # $t23 = 100
32       addi $t24, $zero, 100 # $t24 = 100
33       addi $t25, $zero, 100 # $t25 = 100
34       addi $t26, $zero, 100 # $t26 = 100
35       addi $t27, $zero, 100 # $t27 = 100
36       addi $t28, $zero, 100 # $t28 = 100
37       addi $t29, $zero, 100 # $t29 = 100
38       addi $t30, $zero, 100 # $t30 = 100
39       addi $t31, $zero, 100 # $t31 = 100
40
41       li $t0, 4 # $t0 = 4 for print string
42       li $t1, 0x4e454d4f # print to the log
43       syscall
```

Register dump table:

S	T	A	V	Stack	Log
\$0				10	
\$1				9	
\$2				8	
\$3				7	
\$4				6	
\$5				5	
\$6				4	
\$7				3	
\$8				2	
\$9				1	
\$10				0	
\$11				418	

- **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



The screenshot shows the StackFrame Demo interface. At the top, there are tabs for User, Show, and Go, along with links for ShowFrame Demo, User Guide, and Unit Tests. Below the tabs is a menu bar with File, Addition Calculator, ItInv, Looper, Stack Test, Help, World, Code Gen, Save String, Interactive, Binary, Decimal, and Debug. The main area displays assembly code for a 'Hello World' program. The code includes instructions like ADDI, ADD, SUB, and JAL, along with comments and labels. To the right of the assembly code is a register dump table with columns for S, T, A, V, Stack, and Log. The registers shown are \$0 through \$7, with their values listed in the table.

S	T	A	V	Stack	Log
\$0	10				
\$1	9				
\$2	8				
\$3	7				
\$4	6				
\$5	5				
\$6	807				
\$7	418				

- **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

The screenshot shows the ShowMe Demo interface. At the top, there are tabs for User, ShowMe Demo, Addition, Subtraction, Bitwise, Looper, Stack Test, Hello World, Code Gen, Save String, Interactive, Binary, Decimal, Debug, and Help. Below the tabs is a toolbar with icons for Addition, Subtraction, Bitwise, Looper, Stack Test, and Hello World. A status bar at the bottom says "User Guide | Unit Tests | Docs".  
  
The main area displays assembly code and a register dump. The assembly code is:

```
1 # Shows "Hello world" at the top of the stack
2 .data
3 hello: .asciiz "Hello world"
4 .text
5 addi $t0, $zero, 101 # $t0 = 101
6 addi $t1, $zero, 100 # $t1 = 100
7 addi $t2, $zero, 100 # $t2 = 100
8 addi $t3, $zero, 100 # $t3 = 100
9 addi $t4, $zero, 100 # $t4 = 100
10 addi $t5, $zero, 100 # $t5 = 100
11 addi $t6, $zero, 100 # $t6 = 100
12 addi $t7, $zero, 100 # $t7 = 100
13 addi $t8, $zero, 100 # $t8 = 100
14 addi $t9, $zero, 100 # $t9 = 100
15 addi $t10, $zero, 100 # $t10 = 100
16 addi $t11, $zero, 100 # $t11 = 100
17 addi $t12, $zero, 100 # $t12 = 100
18 addi $t13, $zero, 100 # $t13 = 100
19 addi $t14, $zero, 100 # $t14 = 100
20 addi $t15, $zero, 100 # $t15 = 100
21 addi $t16, $zero, 100 # $t16 = 100
22 addi $t17, $zero, 100 # $t17 = 100
23 addi $t18, $zero, 100 # $t18 = 100
24 addi $t19, $zero, 100 # $t19 = 100
25 addi $t20, $zero, 100 # $t20 = 100
26 addi $t21, $zero, 100 # $t21 = 100
27 addi $t22, $zero, 0 # (call)
28 addi $t23, $zero, 0 # (return)
29 addi $t24, $zero, 4 # $t24 = 4 for print string
30 addi $t25, $zero, 0 # $t25 = 0
31 addi $t26, $zero, 0 # $t26 = 0
32 syscall
```

  
To the right of the assembly code is a register dump table with columns S, T, A, V, Stack, and Log. The register values are:

S	T	A	V	Stack	Log
\$0	10				
\$1	9				
\$2	8				
\$3	7				
\$4	6				
\$5	5				
\$6	807				
\$7	418				

- **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

The screenshot shows the StackFrame Demo application interface. At the top, there are tabs for User, ShowFrame Demo, Addition, Subtraction, Bitwise, Looper, Stack Test, Hello World, Interactive, Direct Decimal, Decimal Binary, and Debug. Below the tabs, there is a text area containing assembly code:

```
# Shows "Hello world" at the top of the stack
1.    .data
2.    msg: .asciiz "Hello world"
3.    .text
4.    .globl _start
5.    _start:
6.        # move $msg to $t0
7.        li $t0, msg
8.        # add $t0, $t0, $t0
9.        add $t0, $t0, $t0
10.       # move $t0 to $s0
11.       sw $t0, 12($sp)
12.       # move $t0 to $t1
13.       li $t1, 12
14.       # move $t1 to $t0
15.       sw $t1, ($t0)
16.       # move $t0 to $t1
17.       li $t1, 12
18.       # move $t1 to $t0
19.       sw $t1, ($t0)
20.       # move $t0 to $t1
21.       li $t1, 12
22.       # move $t1 to $t0
23.       sw $t1, ($t0)
24.       # move $t0 to $t1
25.       li $t1, 12
26.       # move $t1 to $t0
27.       sw $t1, ($t0)
28.       # move $t0 to $t1
29.       li $t1, 12
30.       # move $t1 to $t0
31.       sw $t1, ($t0)
32.       # print to the log
33.       syscall
```

Below the assembly code, there is a table titled "Registers" with columns S, T, A, V, Stack, and Log. The table shows the following register values:

	S	T	A	V	Stack	Log
\$0	10					
\$1	9					
\$2	8					
\$3	7					
\$4	6					
\$5	5					
\$6	807					
\$7	418					

- **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)

# In Pairs or Triples:

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      # print to the log
32 syscall
```

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				

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

# WeMIPS

User | 3 | Oct | ShowHide Device | User Guide | Unit Tests | Docs

Addition Doubler | Stax | Looper | Stack Test | Hello World | Code Gen Save String | Interactive | Binary2 Decimal | Decimal2 Binary | Debug

```
# Store 'Hello world!' at the top of the stack
1 #!/usr/bin/we MIPS
2 ADDI $t0, $zero, 72 # N
3 ADDI $t1, $zero, 101 # e
4 ADDI $t2, $zero, 101 # m
5 ADDI $t3, $zero, 101 # l
6 ADDI $t4, $zero, 101 # o
7 ADDI $t5, $zero, 108 # w
8 ADDI $t6, $zero, 108 # r
9 ADDI $t7, $zero, 108 # l
10 ADDI $t8, $zero, 108 # d
11 ADDI $t9, $zero, 101 # n
12 ADDI $t10, $zero, 41 # p
13 ADDI $t11, $zero, 128 # ! (paren)
14 ADDI $t12, $zero, 128 # ) (close)
15 ADDI $t13, $zero, 119 # [
16 ADDI $t14, $zero, 119 # ]
17 ADDI $t15, $zero, 113 # {
18 ADDI $t16, $zero, 113 # }
19 ADDI $t17, $zero, 114 # r
20 ADDI $t18, $zero, 114 # c
21 ADDI $t19, $zero, 108 # i
22 ADDI $t20, $zero, 108 # o
23 ADDI $t21, $zero, 108 # n
24 ADDI $t22, $zero, 109 # d
25 ADDI $t23, $zero, 107 # p
26 ADDI $t24, $zero, 33 # !
27 ADDI $t25, $zero, 111 # (
28 ADDI $t26, $zero, 0 # (null)
29 ADDI $t27, $zero, 127 # )
30 ADDI $t28, $zero, 127 # )
31 ADDI $t29, $zero, 4 # 4 is for print string
32 ADDI $t30, $zero, 0 # point to the log
33 syscall
```

Step	Run	<input checked="" type="checkbox"/> Enable auto switching			
S	T	A	V	Stack	Log
s0	10				
s1	9				
s2	9				
s3	22				
s4	696				
s5	976				
s6	977				
s7	419				

(Demo with WeMIPS)

# Today's Topics



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

# 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.



A screenshot of a hex editor application. The left pane shows a list of memory pages, each containing a series of memory addresses and their corresponding byte values. The right pane is a detailed view of a specific page, showing memory addresses from 0x00000000 to 0x0000000F. The bytes displayed are: 48 45 4C 4C 4D 4E 4F 4F 4A 4B 4C 4D 4E 4F 4F 4A 4B 4C. Below the address column, there is a column labeled 'Label' which contains the labels 'start', 'loop', and 'end'. The bottom of the window has a status bar with the text 'File Edit View Insert Options Window Help'.

# 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

```
User | 3 | Dat Show White Device  
Addition Doubler | Stax | Looper | Stack Test | Hello World |  
Code Gen Save String | Interactive | Binary2 Decimal | Decimal2 Binary |  
Debug  
  
# Store 'Hello world!' at the top of the stack  
0: ADDI $t0, $zero, 72 # $H  
1: ADDI $t1, $zero, 101 # $e  
2: ADDI $t2, $zero, 101 # $e  
3: ADDI $t3, $zero, 108 # l  
4: ADDI $t4, $zero, 108 # l  
5: ADDI $t5, $zero, 108 # i  
6: ADDI $t6, $zero, 108 # n  
7: ADDI $t7, $zero, 108 # n  
8: ADDI $t8, $zero, 108 # o  
9: ADDI $t9, $zero, 108 # n  
10: ADDI $t10, $zero, 33 # ! (space)  
11: ADDI $t11, $zero, 41 # s  
12: ADDI $t12, $zero, 41 # s  
13: ADDI $t13, $zero, 113 # l  
14: ADDI $t14, $zero, 113 # l  
15: ADDI $t15, $zero, 113 # l  
16: ADDI $t16, $zero, 113 # l  
17: ADDI $t17, $zero, 113 # l  
18: ADDI $t18, $zero, 114 # r  
19: ADDI $t19, $zero, 114 # r  
20: ADDI $t20, $zero, 108 # i  
21: ADDI $t21, $zero, 108 # i  
22: ADDI $t22, $zero, 108 # i  
23: ADDI $t23, $zero, 108 # d  
24: ADDI $t24, $zero, 101 # e  
25: ADDI $t25, $zero, 33 # !  
26: ADDI $t26, $zero, 41 # s  
27: ADDI $t27, $zero, 0 # (null)  
28: ADDI $t28, $zero, 114 # r  
29: ADDI $t29, $zero, 114 # r  
30: ADDI $t30, $zero, 4 # 4 is for print string  
31: ADDI $t31, $zero, 0 # point to the log  
32: syscall
```

Step	Run	<input checked="" type="checkbox"/> Enable auto switching			
S	T	A	V	Stack	Log
s0				10	
s1				9	
s2				9	
s3				22	
s4				696	
s5				976	
s6				977	
s7				418	

(Demo with WeMIPS)

# Today's Topics



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

# In Pairs or Triples:

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.')
```

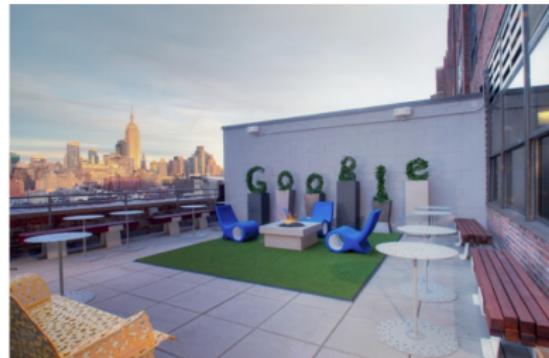
- Example of **linear search**.
- Start at the beginning of the list.
- Look at each item, one-by-one.
- Stopping, when found, or the end of list is reached.

# Today's Topics



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

# CS Survey Talk

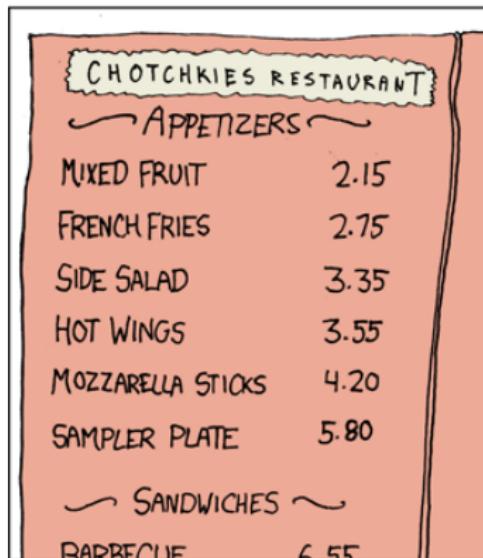


[careers.google.com](https://careers.google.com)

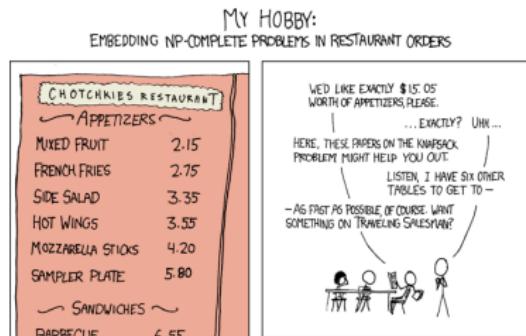
Anna Whitney  
(Google)

# Design Challenge

## MY HOBBY: EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



# Design Challenge



- Possible solutions:

# Design Challenge

MY HOBBY:  
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



- Possible solutions:

- ▶ 7 orders of mixed fruit, or

# Design Challenge

MY HOBBY:  
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



- Possible solutions:

- ▶ 7 orders of mixed fruit, or
- ▶ 2 orders hot wings, 1 order mixed fruit, and 1 sampler plate.

# Design Challenge

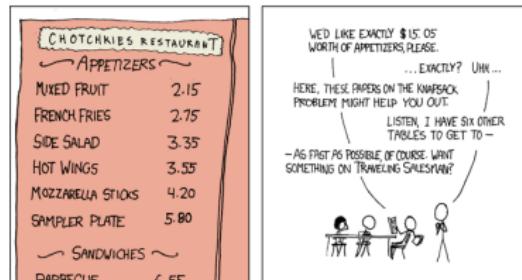
MY HOBBY:  
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



- Possible solutions:
  - ▶ 7 orders of mixed fruit, or
  - ▶ 2 orders hot wings, 1 order mixed fruit, and 1 sampler plate.
- **Input:** List of items with prices and amount to be spent.

# Design Challenge

MY HOBBY:  
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



- Possible solutions:
  - ▶ 7 orders of mixed fruit, or
  - ▶ 2 orders hot wings, 1 order mixed fruit, and 1 sampler plate.
- **Input:** List of items with prices and amount to be spent.
- **Output:** An order that totals to the amount or empty list if none.

# Design Challenge

MY HOBBY:  
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



- Possible solutions:
  - ▶ 7 orders of mixed fruit, or
  - ▶ 2 orders hot wings, 1 order mixed fruit, and 1 sampler plate.
- **Input:** List of items with prices and amount to be spent.
- **Output:** An order that totals to the amount or empty list if none.
- Possible algorithms: For each item on the list, divide total by price. If no remainder, return a list of that item. Repeat with two items, trying 1 of the first, 2 of the first, etc. Repeat with three items, etc.

# Design Challenge

MY HOBBY:  
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



- Possible solutions:
  - ▶ 7 orders of mixed fruit, or
  - ▶ 2 orders hot wings, 1 order mixed fruit, and 1 sampler plate.
- **Input:** List of items with prices and amount to be spent.
- **Output:** An order that totals to the amount or empty list if none.
- Possible algorithms: For each item on the list, divide total by price. If no remainder, return a list of that item. Repeat with two items, trying 1 of the first, 2 of the first, etc. Repeat with three items, etc.
- “NP-Complete” problem: possible answers can be checked quickly, but not known how to compute quickly.

# Recap

- On lecture slip, write down a topic you wish we had spent more time (and why).



# Recap



- On lecture slip, write down a topic you wish we had spent more time (and why).
- Searching through data is a common task— built-in functions and standard design patterns for this.

# Recap



- On lecture slip, write down a topic you wish we had spent more time (and why).
- 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.

# Recap



- On lecture slip, write down a topic you wish we had spent more time (and why).
- 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.
- Pass your lecture slips to the aisles for the UTAs to collect.

# Final Overview: Top-Down Design & APIs

For each question, write **only the function header (name & inputs) and return values** (often called the Application Programming Interface (API)):

# Final Overview: Top-Down Design & APIs

For each question, write **only the function header (name & inputs) and return values** (often called the Application Programming Interface (API)):

- Write a function that takes a weight in kilograms and returns the weight in pounds.
- Write a function that takes a string and returns its length.
- Write a function that, given a DataFrame, returns the minimal value in the first column.
- Write a function that takes a whole number and returns the corresponding binary number as a string.
- Write a function that computes the total monthly payment when given the initial loan amount, annual interest rate, number of years of the loan.

# Final Overview: Top-Down Design & APIs

For each question, write **only the function header (name & inputs) and return values** (often called the Application Programming Interface (API)):

- Write a function that takes a weight in kilograms and returns the weight in pounds.
- Write a function that takes a string and returns its length.
- Write a function that, given a DataFrame, returns the minimal value in the first column.
- Write a function that takes a whole number and returns the corresponding binary number as a string.
- Write a function that computes the total monthly payment when given the initial loan amount, annual interest rate, number of years of the loan.

*(Hint: highlight key words, make list of inputs, list of outputs, then put together.)*

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a weight in kilograms and returns the weight in pounds.

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a weight in kilograms and returns the weight in pounds.

```
def kg2lbs(kg):  
    ...  
    return(lbs)
```

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a weight in kilograms and returns the weight in pounds.

```
def kg2lbs(kg)
    lbs = kg * 2.2
    return(lbs)
```

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a string and returns its length.

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a string and returns its length.

```
def sLength(str):  
    ...  
    return(length)
```

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a string and returns its length.

```
def sLength(str):  
    length = len(str)  
    return(length)
```

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that, given a DataFrame, returns the minimal value in the “Manhattan” column.

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that, given a DataFrame, returns the minimal value in the “Manhattan” column.

```
def getMin(df):  
    ...  
    return(min)
```

# Final Overview

For each question below, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that, given a DataFrame, returns the minimal value in the “Manhattan” column.

```
def getMin(df):
    min = df['Manhattan'].min()
    return(min)
```

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a whole number and returns the corresponding binary number as a string.

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a whole number and returns the corresponding binary number as a string.

```
def num2bin(num):  
    ...  
    return(bin)
```

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that takes a whole number and returns the corresponding binary number as a string.

```
def num2bin(num):  
    binStr = ""  
    while (num > 0):  
        #Divide by 2, and add the remainder to the string  
        r = num %2  
        binString = str(r) + binStr  
        num = num / 2  
    return(binStr)
```

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that computes the total monthly payment when given the initial loan amount, annual interest rate, number of years of the loan.

# Final Overview

For each question, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that computes the total monthly payment when given the initial loan amount, annual interest rate, number of years of the loan.

```
def computePayment(loan,rate,year):  
    ....  
    return(payment)
```

# Final Overview

For each question below, write the function header (name & inputs) and return values (often called the Application Programming Interface (API)):

- Write a function that computes the total monthly payment when given the initial loan amount, annual interest rate, number of years of the loan.

```
def computePayment(loan,rate,year):  
    (Some formula for payment)  
    return(payment)
```

# Writing Boards



- Return writing boards as you leave...